

**DIRECTORATE OF DISTANCE EDUCATION**

**UNIVERSITY OF NORTH BENGAL**

**MASTER OF SCIENCES- MATHEMATICS**

**SEMESTER -IV**

**NUMERICAL PROBLEM SOLVING BY  
COMPUTER PROGRAMMING (THEORY)**

**DEMATH4SCORE2**

**BLOCK-2**

---

## UNIVERSITY OF NORTH BENGAL

Postal Address:

The Registrar,

University of North Bengal,

Raja Rammohunpur,

P.O.-N.B.U.,Dist-Darjeeling,

West Bengal, Pin-734013,

India.

Phone: ( O ) +91 0353-2776331/2699008

Fax:( 0353 ) 2776313, 2699001

Email: regnbu@sancharnet.in ; regnbu@nbu.ac.in

Website: www.nbu.ac.in

First Published in 2019



All rights reserved. No Part of this book may be reproduced or transmitted, in any form or by any means, without permission in writing from University of North Bengal. Any person who does any unauthorised act in relation to this book may be liable to criminal prosecution and civil claims for damages. This book is meant for educational and learning purpose. The authors of the book has/have taken all reasonable care to ensure that the contents of the book do not violate any existing copyright or other intellectual property rights of any person in any manner whatsoever. In the even the Authors has/ have been unable to track any source and if any copyright has been inadvertently infringed, please notify the publisher in writing for corrective action.

## **FOREWORD**

The Self Learning Material (SLM) is written with the aim of providing simple and organized study content to all the learners. The SLMs are prepared on the framework of being mutually cohesive, internally consistent and structured as per the university's syllabi. It is a humble attempt to give glimpses of the various approaches and dimensions to the topic of study and to kindle the learner's interest to the subject

We have tried to put together information from various sources into this book that has been written in an engaging style with interesting and relevant examples. It introduces you to the insights of subject concepts and theories and presents them in a way that is easy to understand and comprehend.

We always believe in continuous improvement and would periodically update the content in the very interest of the learners. It may be added that despite enormous efforts and coordination, there is every possibility for some omission or inadequacy in few areas or topics, which would definitely be rectified in future.

We hope you enjoy learning from this book and the experience truly enrich your learning and help you to advance in your career and future endeavours.

---

---

# **NUMERICAL PROBLEM SOLVING BY COMPUTER PROGRAMMING**

---

## **BLOCK-1**

Unit-1: Computer Programming

Unit-2: Introduction To C

Unit-3: Elements Of C -I

Unit-4: Elements Of C -Ii

Unit-5: Expression And 'If' Statement In C

Unit-6: Branching And Looping Statement In C

Unit-7: Designing Structured Programs In C

## **BLOCK-2**

<b>UNIT-8 Inter Function Communication And Recursive Function In C .....</b>	<b>6</b>
<b>UNIT-9 ARRAYS .....</b>	<b>25</b>
<b>UNIT-10 Pointers In C.....</b>	<b>41</b>
<b>UNIT-11 Dynamic Memory Allocation And Strings In C.....</b>	<b>58</b>
<b>UNIT-12 Structure In C And File Handling.....</b>	<b>76</b>
<b>UNIT-13 File Handling Function And Error Handling In C.....</b>	<b>92</b>
<b>UNIT 14: APPLICATION OF C IN NUMERICAL ANALYSIS ..</b>	<b>108</b>

---

# **BLOCK-2 NUMERICAL PROBLEM SOLVING BY COMPUTER PROGRAMMING**

---

**Programming** is the process of taking an algorithm and encoding it into C was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C. Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is reliable, simple and easy to use. often heard today is – “C has been already superceded by languages like C++, C# and Java.

---

# **UNIT-8 INTER FUNCTION COMMUNICATION AND RECURSIVE FUNCTION IN C**

---

## **STRUCTURE**

8.0 Objectives

8.1 Introduction

8.2 Inter Function Communication in C

8.2.1 Downward Communication

8.2.2 Upward Communication

8.2.3 Bi-directional Communication

8.3 Standard Functions in C

8.4 Scope of Variable in C

8.4.1 Before the function definition (Global Declaration)

8.4.2 Inside the function or block (Local Declaration)

8.4.3 In the function definition parameters (Formal Parameters)

8.5 Recursive Function in C

8.6 Type Qualifiers in C

8.7 Preprocessor Command in C

8.8 Lets Sum Up

8.9 Keywords

8.10 Questions for Review

8.11 Suggested Reading and References

8.12 Answers to Check your Progress

---

## **8.0 OBJECTIVES**

---

Understands the Inter Function Communication in C

Comprehend Standard Functions & scope of variable in C

Enumerate Recursive Function in C

Understands Type Qualifiers and Preprocessor Command in C

---

## **8.1 INTRODUCTION**

---

A computer program cannot handle all the tasks by itself. It requests other program like entities called functions in C. We pass information to the function called arguments which specified when the function is called. A function either can return a value or returns nothing. Function is a subprogram that helps reduce coding.

---

## **8.2 INTER FUNCTION COMMUNICATION IN C**

---

When a function gets executed in the program, the execution control is transferred from calling a function to called function and executes function definition, and finally comes back to the calling function. In this process, both calling and called functions have to communicate with each other to exchange information. The process of exchanging information between calling and called functions is called inter-function communication.

In C, the inter function communication is classified as follows...

- **Downward Communication**
- **Upward Communication**
- **Bi-directional Communication**

### **8.2.1 DOWNWARD COMMUNICATION**

In this type of inter function communication, the data is transferred from calling function to called function but not from called function to calling function. The functions with parameters and without return value are considered under downward communication. In the case of downward communication, the execution control jumps from calling function to

## Notes

called function along with parameters and executes the function definition, and finally comes back to the calling function without any return value. For example consider the following program...

### Example Program

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

void main(){
    int num1, num2 ;
    void addition(int, int) ; // function declaration
    clrscr() ;
    num1 = 10 ;
    num2 = 20 ;

    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
    addition(num1, num2) ; // calling function

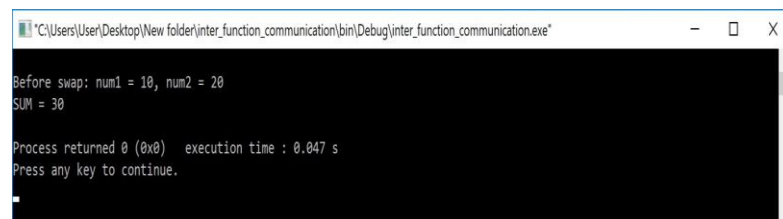
    getch() ;
}

void addition(int a, int b) // called function
{

    printf("SUM = %d", a+b) ;

}
```

### Output:



```
"C:\Users\User\Desktop\New folder\inter_function_communication\bin\Debug\inter_function_communication.exe"
Before swap: num1 = 10, num2 = 20
SUM = 30

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

## 8.2.2 UPWARD COMMUNICATION

In this type of inter-function communication, the data is transferred from called function to calling-function but not from calling-function to called-function. The functions without parameters and with return value are considered under upward communication. In the case of upward communication, the execution control jumps from calling-function to



called-function without parameters and executes the function definition, and finally comes back to the calling function along with a return value. For example, consider the following program...

### Example Program

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

void main(){
    int result ;
    int addition() // function declaration
    clrscr() ;

    result = addition() // calling function

    printf("SUM = %d", result) ;
    getch() ;
}
int addition() // called function
{
    int num1, num2 ;
    num1 = 10;
    num2 = 20;
    return (num1+num2);
}
```

### Output:

```
"C:\Users\User\Desktop\New folder\inter_function_communication\bin\Debug\inter_function_communication.exe"
SUM = 30
Process returned 0 (0x0) execution time : 0.047 s
Press any key to continue.
```

### 8.2.3 BI - DIRECTIONAL COMMUNICATION

In this type of inter-function communication, the data is transferred from calling-function to called function and also from called function to calling-function. The functions with parameters and with return value are considered under bi-directional communication. In the case of bi-directional communication, the execution control jumps from calling-function to called function along with parameters and executes the function definition and finally comes back to the calling function along with a return value. For example, consider the following program...

### Example Program

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

void main(){
    int num1, num2, result ;
    int addition(int, int) ; // function declaration
    clrscr() ;

    num1 = 10 ;
    num2 = 20 ;

    result = addition(num1, num2) ; // calling function

    printf("SUM = %d", result) ;
    getch() ;
}
int addition(int a, int b) // called function
{
    return (a+b) ;
}
```

### Output:

```
C:\Users\User\Desktop\New folder\inter_function_communication\bin\Debug\inter_function_communication.exe
SUM = 30
Process returned 0 (0x0) execution time : 0.047 s
Press any key to continue.
```

---

## 8.3 STANDARD FUNCTIONS IN C

---

The standard functions are built-in functions. In C programming language, the standard functions are declared in header files and defined in .dll files. In simple words, the standard functions can be defined as "the readymade functions defined by the system to make coding more easy". The standard functions are also called as **library functions** or **pre-defined functions**.

In C when we use standard functions, we must include the respective header file using **#include** statement. For example, the function **printf()** is defined in header file **stdio.h** (Standard Input Output

header file). When we use **printf()** in our program, we must include **stdio.h** header file using **#include<stdio.h>** statement.

C Programming Language provides the following header files with standard functions.

Header File	Purpose	Example Functions
<b>stdio.h</b>	Provides functions to perform standard I/O operations	printf(), scanf()
<b>conio.h</b>	Provides functions to perform console I/O operations	clrscr(), getch()
<b>math.h</b>	Provides functions to perform mathematical operations	sqrt(), pow()
<b>string.h</b>	Provides functions to handle string data values	strlen(), strcpy()
<b>stdlib.h</b>	Provides functions to perform general functions	calloc(), malloc()
<b>time.h</b>	Provides functions to perform operations on time and date	time(), localtime()
<b>ctype.h</b>	Provides functions to perform - testing and mapping of character data values	isalpha(), islower()
<b>setjmp.h</b>	Provides functions that are used in function calls	setjump(), longjump()
<b>signal.h</b>	Provides functions to handle signals during program execution	signal(), raise()
<b>assert.h</b>	Provides Macro that is used to verify assumptions made by the program	assert()
<b>locale.h</b>	Defines the location specific settings such as date formats and currency symbols	setlocale()
<b>stdarg.h</b>	Used to get the arguments in a function if the arguments are not specified by the function	va_start(), va_end(), va_arg()

## Notes

<b>errno.h</b>	Provides macros to handle the system calls	Error, errno
<b>graphics.h</b>	Provides functions to draw graphics.	circle(), rectangle()
<b>float.h</b>	Provides constants related to floating point data values	
<b>stddef.h</b>	Defines various variable types	
<b>limits.h</b>	Defines the maximum and minimum values of various variable types like char, int and long	

---

## 8.4 SCOPE OF VARIABLE IN C

---

When we declare a variable in a program, it can not be accessed against the scope rules. Variables can be accessed based on their scope. The scope of a variable decides the portion of a program in which the variable can be accessed. The scope of the variable is defined as follows...

**Scope of a variable is the portion of the program where a defined variable can be accessed.**

The variable scope defines the visibility of variable in the program. Scope of a variable depends on the position of variable declaration. In C programming language, a variable can be declared in three different positions and they are as follows...

- **Before the function definition (Global Declaration)**
- **Inside the function or block (Local Declaration)**
- **In the function definition parameters (Formal Parameters)**

### 8.4.1 BEFORE THE FUNCTION DEFINITION (GLOBAL DECLARATION)

Declaring a variable before the function definition (outside the function definition) is called **global declaration**. The variable declared using global declaration is called **global variable**. The global variable can be accessed by all the functions that are defined after the global declaration.

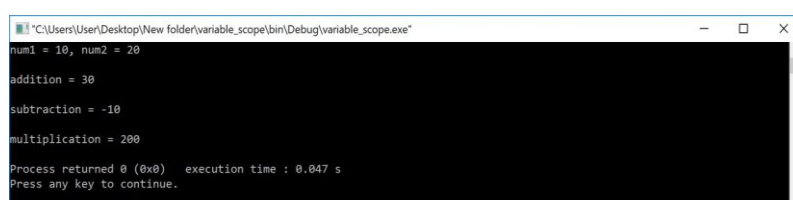
That means the global variable can be accessed anywhere in the program after its declaration. The global variable scope is said to be **file scope**.

### Example Program

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

int num1, num2 ;
void main(){
    void addition() ;
    void subtraction() ;
    void multiplication() ;
    clrscr() ;
    num1 = 10 ;
    num2 = 20 ;
    printf("num1 = %d, num2 = %d", num1, num2) ;
    addition() ;
    subtraction() ;
    multiplication() ;
    getch() ;
}
void addition()
{
    int result ;
    result = num1 + num2 ;
    printf("\naddition = %d", result) ;
}
void subtraction()
{
    int result ;
    result = num1 - num2 ;
    printf("\nsubtraction = %d", result) ;
}
void multiplication()
{
    int result ;
    result = num1 * num2 ;
    printf("\nmultiplication = %d", result) ;
}
```

### Output:



```
C:\Users\User\Desktop\New folder\variable_scope\bin\Debug\variable_scope.exe
num1 = 10, num2 = 20
addition = 30
subtraction = -10
multiplication = 200
Process returned 0 (0x0) execution time : 0.047 s
Press any key to continue.
```

## Notes

In the above example program, the variables **num1** and **num2** are declared as global variables. They are declared before the `main()` function. So, they can be accessed by function `main()` and other functions that are defined after `main()`. In the above example, the functions `main()`, `addition()`, `subtraction()` and `multiplication()` can access the variables `num1` and `num2`.

### 8.4.2 INSIDE THE FUNCTION OR BLOCK (LOCAL DECLARATION)


Declaring a variable inside the function or block is called **local declaration**. The variable declared using local declaration is called **local variable**. The local variable can be accessed only by the function or block in which it is declared. That means the local variable can be accessed only inside the function or block in which it is declared.

#### Example Program

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

void main(){
    void addition() ;
    int num1, num2 ;
    clrscr() ;
    num1 = 10 ;
    num2 = 20 ;
    printf("num1 = %d, num2 = %d", num1, num2) ;
    addition() ;
    getch() ;
}
void addition()
{
    int sumResult ;
    sumResult = num1 + num2 ;
    printf("\naddition = %d", sumResult) ;
}
```

#### Output:



```
C:\Users\User\Desktop\New folder\variable_scope\bin\Debug\variable_scope.exe
num1 = 10, num2 = 20
addition = 0
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

The above example program shows an **error** because, the variables num1 and num2 are declared inside the function main(). So, they can be used only inside main() function and not in addition() function.

### 8.4.3 IN THE FUNCTION DEFINITION PARAMETERS (FORMAL PARAMETERS)

The variables declared in function definition as parameters have a local variable scope. These variables behave like local variables in the function. They can be accessed inside the function but not outside the function.

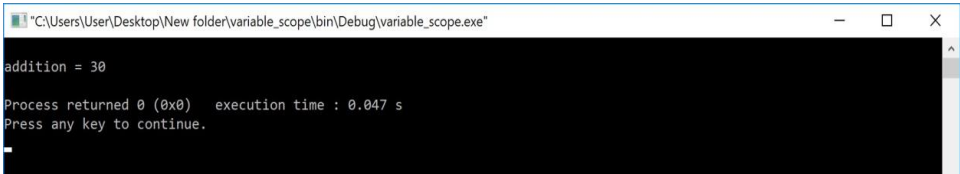
#### Example Program

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

void main(){
    void addition(int, int) ;
    int num1, num2 ;
    clrscr() ;
    num1 = 10 ;
    num2 = 20 ;
    addition(num1, num2) ;
    getch() ;
}

void addition(int a, int b)
{
    int sumResult ;
    sumResult = a + b ;
    printf("\naddition = %d", sumResult) ;
}
```

#### Output:



```
"C:\Users\User\Desktop\New folder\variable_scope\bin\Debug\variable_scope.exe"
addition = 30
Process returned 0 (0x0) execution time : 0.047 s
Press any key to continue.
```

## Notes

In the above example program, the variables a and b are declared in function definition as parameters. So, they can be used only inside the addition () function.

### Check your Progress-1

1. What is Downward Communication?

---

---

---

2. Define Scope of Variable

---

---

---

3. What is local variable?

---

---

---

---

## 8.5 RECURSIVE FUNCTIONS IN C

---

In C programming language, function calls can be made from the main() function, other functions or from the same function itself. The recursive function is defined as follows...

**A function called by itself is called recursive function.**

The recursive functions should be used very carefully because, when a function called by itself it enters into the infinite loop. And when a function enters into the infinite loop, the function execution never gets completed. We should define the condition to exit from the function call so that the recursive function gets terminated.

When a function is called by itself, the first call remains under execution till the last call gets invoked. Every time when a function call is invoked, the function returns the execution control to the previous function call.



## Example Program

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

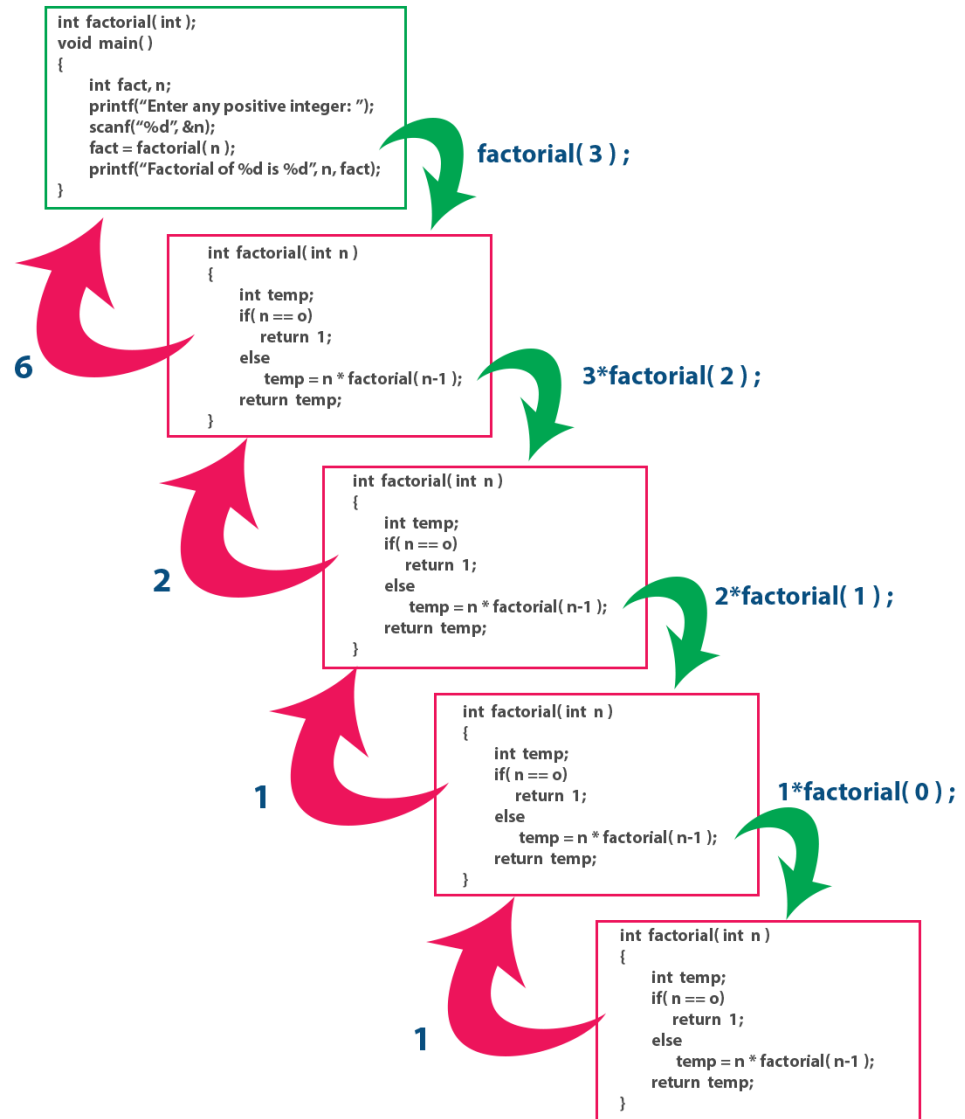
int factorial( int ) ;
int main()
{
    int fact, n ;
    printf("Enter any positive integer: ") ;
    scanf("%d", &n) ;
    fact = factorial( n ) ;
    printf("\nFactorial of %d is %d\n", n, fact) ;
    return 0;
}
int factorial( int n )
{
    int temp ;
    if( n == 0)
        return 1 ;
    else
        temp = n * factorial( n-1 ) ; // recursive function call
    return temp ;
}
```

### Output:

```
"C:\Users\User\Desktop\New folder\recursive_function\bin\Debug\recursive_function.exe"
Enter any positive integer: 5
Factorial of 5 is 120
Process returned 0 (0x0) execution time : 2.235 s
Press any key to continue.
```

In the above example program, the **factorial()** function call is initiated from `main()` function with the value 3. Inside the `factorial()` function, the function calls `factorial(2)`, `factorial(1)` and `factorial(0)` are called recursively. In this program execution process, the function call `factorial(3)` remains under execution till the execution of function calls `factorial(2)`, `factorial(1)` and `factorial(0)` gets completed. Similarly the function call `factorial(2)` remains under execution till the execution of function calls `factorial(1)` and `factorial(0)` gets completed. In the same way the function call `factorial(1)` remains under execution till the execution of function call `factorial(0)` gets completed. The complete execution process of the above program is shown in the following figure...

## Notes



---

## 8.6 TYPE QUALIFIERS IN C

---

In C programming language, type qualifiers are the keywords used to modify the properties of variables. Using type qualifiers, we can change the properties of variables. The C programming language provides two type qualifiers and they are as follows...

- **const**
- **volatile**

### 8.6.1 const TYPE QUALIFIER IN C

The **const** type qualifier is used to create constant variables. When a variable is created with **const** keyword, the value of that variable can't be changed once it is defined. That means once a value is assigned to a constant variable, that value is fixed and cannot be changed throughout the program.

The keyword **const** is used at the time of variable declaration. We use the following syntax to create constant variable using **const** keyword.

```
const datatype variableName ;
```

When a variable is created with **const** keyword it becomes a constant variable. The value of the constant variable can't be changed once it is defined. The following program generates error message because we try to change the value of constant variable **x**.

### Example Program

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

void main(){

    int i = 9 ;
    const int x = 10 ;
    clrscr() ;

    i = 15 ;
    x = 100 ; // creates an error

    printf("i = %d\nx = %d", i, x ) ;

}
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int i = 5 ;
7     const int x = 10 ;
8
9     i = 15 ;
10    x = 100 ; // creates an error
11
12    printf("i = %d\nx = %d", i, x ) ;
13    return 0;
14 }
15
```

Build messages window:

File	Line	Message
C:\Users\user\... 10		error: assignment of read-only variable 'x'

## 8.6.2 volatile TYPE QUALIFIER IN C

The **volatile** type qualifier is used to create variables whose values can't be changed in the program explicitly but can be changed by any external device or hardware.

For example, the variable which is used to store system clock is defined as a volatile variable. The value of this variable is not changed explicitly in the program but is changed by the clock routine of the operating system.

---

## 8.7 PREPROCESSOR COMMANDS IN C

---

In C programming language, preprocessor directive is a step performed before the actual source code compilation. It is not part of the compilation. Preprocessor directives in C programming language are used to define and replace tokens in the text and also used to insert the contents of other files into the source file.

When we try to compile a program, preprocessor commands are executed first and then the program gets compiled.

Every preprocessor command begins with # symbol. We can also create preprocessor commands with parameters.

Following are the preprocessor commands in C programming language...

## #define

#define is used to create symbolic constants (known as macros) in C programming language. This preprocessor command can also be used with parameterized macros.

### Example Program

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

#define PI 3.14

#define SQR(x) x*x      //Parameterized Macro

void main(){

    double radius, area ;
    clrscr() ;

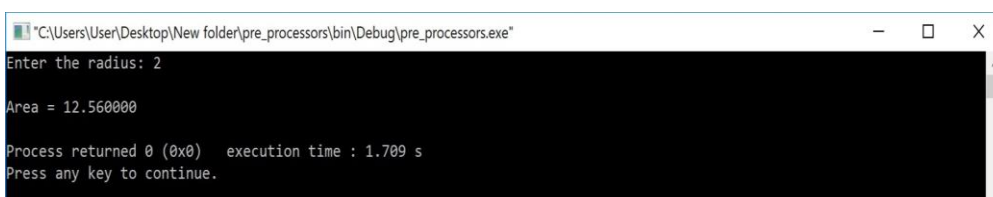
    printf("Enter the radius: ");
    scanf("%ld",&radius);

    area = PI * SQR(radius) ;

    printf("area = %ld",area);

    getch();
}
```

### Output:



```
"C:\Users\User\Desktop\New folder\pre_processors\bin\Debug\pre_processors.exe"
Enter the radius: 2
Area = 12.560000
Process returned 0 (0x0) execution time : 1.709 s
Press any key to continue.
```

## #undef

#undef is used to destroy a macro that was already created using #define.

## #ifdef

#ifdef returns TRUE if the macro is defined and returns FALSE if the macro is not defined.

## Notes

### **#ifndef**

#ifndef returns TRUE if the specified macro is not defined otherwise returns FALSE.

### **#if**

#if uses the value of specified macro for conditional compilation.

### **#else**

#else is an alternative for #if.

### **#elif**

#elif is a #else followed by #if in one statement.

### **#endif**

#endif is used to terminate preprocessor conditional macro.

### **#include**

#include is used to insert specific header file into C program.

### **#error**

#error is used to print error message on stderr.

### **#pragma**

#pragma is used to issue a special command to the compiler.

In C programming language, there are some pre-defined macros and they are as follows...

1. **\_\_DATE\_\_** : The current date as characters in "MMM DD YYYY" format.
2. **\_\_TIME\_\_** : The current time as characters in "HH : MM : SS" format.
3. **\_\_FILE\_\_** : This contains the current file name.
4. **\_\_LINE\_\_** : This contains the current line number.

5. `__STDC__` : Defines 1 when compiler compiles with ANSI Standards.

### Check your Progress-1

4. State recursive function

---



---



---

5. What is const TYPE QUALIFIER IN C

---



---



---



---

## 8.8 LET US SUM UP

---

Using function it becomes easier to write a program and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently.

Separating the code into modular functions also makes the program easier to design and understand.

---

## 8.9 KEYWORDS

---

**Preprocessor macros** - A **predefined macro** is a **macro** that is already understood by the C **pre** processor without the program needing to **define** it

**Character**- The abbreviation **char** is used as a reserved keyword in some programming languages, such as **C**

**Terminate** - bring to an end.

---

## 8.10 QUESTIONS FOR REVIEW

---

1. Brief Inter Function Communication in C
2. Explain Standard Functions in C
3. Explain Recursive Function with example
4. What are preprocessor commands in C? State few examples

---

## 8.11 SUGGESTED READINGS AND REFERENCES

---

1. B. Gottfried: Programming with C , Tata McGraw-Hill Edition 2002.
2. E. Balagurusamy : Programming in ANSI C, Tata Mcgraw Hill - Edition 2002.
3. Brain W. Kernighan & Dennis M. Ritchie, The C Programme Language, 2nd Edition (ANSI features) , Prentice Hall 1989.
4. Let Us C- Y.P. Kanetkar, BPB Publication - 2002.
5. Analysis of Numerical Methods—Isacsons& Keller.
6. Numerical solutions of Ord. Diff. Equations—M K Jain
7. Numerical solutions of Partial Diff. Equations—G D Smith.
8. Programming with C, B. Gottfried, Tata-McGraw Hill
9. Programming with C, K. R. Venugopal and Sudeep R. Prasad, Tata-McGraw Hill

---

## 8.12 ANSWERS TO CHECK YOUR PROGRESS

---

1. Provide explanation with example - 8.2.1
2. Provide explanation with example – 8.4
3. Provide definition – 8.4.2
4. Provide definition – 8.5
5. Provide explanation – 8.6.1



---

# UNIT-9 ARRAYS

---

## STRUCTURE

9.0 Objectives

9.1 Introduction

9.2 Arrays in C

7.2.1 Declaration of an Array

7.2.2 Accessing individual elements of an Array

9.3 Types of Arrays in C

7.3.1 System Defined

7.3.2 User Defined

9.4 Application of Arrays in C

9.5 Let us sum up

9.6 Keywords

9.7 Questions for Review

9.8 Suggested Reading and References

9.9 Answers to Check your Progress

---

## 9.0 OBJECTIVES

---

Understand the concept of Arrays, its types and application in C.

---

## 9.1 INTRODUCTION

---

When we work with a large number of data values we need that any number of different variables. As the number of variables increases, the complexity of the program also increases and so the programmers get confused with the variable names. There may be situations where we

## Notes

need to work with a large number of similar data values. To make this work easier, C programming language provides a concept called "Array".

---

## 9.2 ARRAYS IN C

---

**An array is a special type of variable used to store multiple values of same data type at a time.**

An array can also be defined as follows...

**An array is a collection of similar data items stored in continuous memory locations with single name.**

### 9.2.1 DECLARATION OF AN ARRAY

In C programming language, when we want to create an array we must know the datatype of values to be stored in that array and also the number of values to be stored in that array.

We use the following general syntax to create an array...

```
datatype arrayName [ size ] ;
```

Syntax for creating an array with size and initial values

```
datatype arrayName [ size ] = {value1, value2,  
...};
```

Syntax for creating an array without size and with initial values

```
datatype arrayName [ ] = {value1, value2, ...};
```

In the above syntax, the **datatype** specifies the type of values we store in that array and **size** specifies the maximum number of values that can be stored in that array.

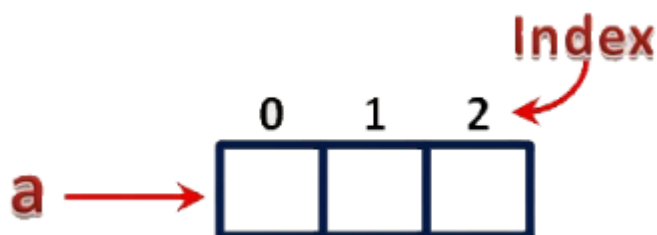
### Example Code

```
int a [3] ;
```

Here, the compiler allocates 6 bytes of contiguous memory locations with a single name 'a' and tells the compiler to store three different integer values (each in 2 bytes of memory) into that 6 bytes of memory. For the above declaration, the memory is organized as follows...



In the above memory allocation, all the three memory locations have a common name 'a'. So accessing individual memory location is not possible directly. Hence compiler not only allocates the memory but also assigns a numerical reference value to every individual memory location of an array. This reference number is called "Index" or "subscript" or "indices". Index values for the above example are as follows...



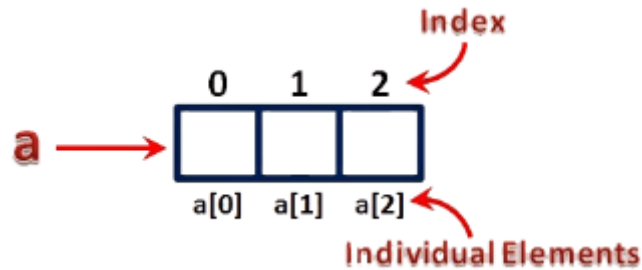
### 9.2.2 ACCESSING INDIVIDUAL ELEMENTS OF AN ARRAY

The individual elements of an array are identified using the combination of 'arrayName' and 'indexValue'. We use the following general syntax to access individual elements of an array...

## Notes

`arrayName [ indexValue ] ;`

For the above example the individual elements can be denoted as follows...

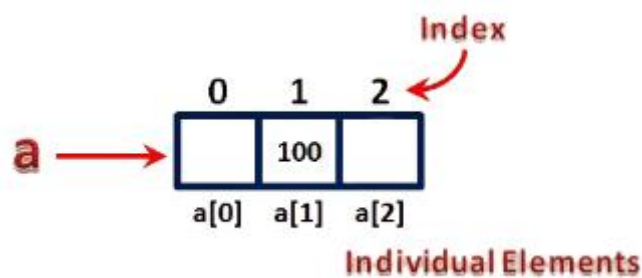


For example, if we want to assign a value to the second memory location of above array 'a', we use the following statement...

### Example Code

```
a [1] = 100 ;
```

The result of the above assignment statement is as follows...



---

## 9.3 TYPES OF ARRAYS IN C

---

In C programming language, arrays are classified into **two types**. They are as follows...

### 1. Single Dimensional Array / One Dimensional Array

## 2. Multi Dimensional Array

### 9.3.1 SINGLE DIMENSIONAL ARRAY

In C programming language, single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array, data is stored in linear form. Single dimensional arrays are also called as **one-dimensional arrays**, **Linear Arrays** or simply **1-D Arrays**.

#### Declaration of Single Dimensional Array

We use the following general syntax for declaring a single dimensional array...

```
datatype arrayName [ size ] ;
```

#### Example Code

```
int rollNumbers [60] ;
```

The above declaration of single dimensional array reserves 60 continuous memory locations of 2 bytes each with the name **rollNumbers** and tells the compiler to allow only integer values into those memory locations.

#### Initialization of Single Dimensional Array

We use the following general syntax for declaring and initializing a single dimensional array with size and initial values.

Example Code

```
datatype arrayName [ size ] = {value1, value2,  
...} ;
```

```
int marks [6] = { 89, 90, 76, 78, 98, 86 } ;
```

## Notes

The above declaration of single dimensional array reserves 6 contiguous memory locations of 2 bytes each with the name **marks** and initializes with value 89 in first memory location, 90 in second memory location, 76 in third memory location, 78 in fourth memory location, 98 in fifth memory location and 86 in sixth memory location.

We can also use the following general syntax to initialize a single dimensional array without specifying size and with initial values...

```
datatype arrayName [ ] = {value1, value2, ...};
```

The array must be initialized if it is created without specifying any size. In this case, the size of the array is decided based on the number of values initialized.

### Example Code

```
int marks [ ] = { 89, 90, 76, 78, 98, 86 } ;
```

```
char studentName [ ] = "btechsmartclass" ;
```

In the above example declaration, size of the array '**marks**' is **6** and the size of the array '**studentName**' is **16**. This is because in case of character array, compiler stores one extra character called **\0** (NULL) at the end.

### **Accessing Elements of Single Dimensional Array**

In C programming language, to access the elements of single dimensional array we use array name followed by index value of the element that to be accessed. Here the index value must be enclosed in square braces. **Index** value of an element in an array is the reference number given to each element at the time of memory allocation. The index value of single dimensional array starts with zero (0) for first element and incremented by one for each element. The index value in an array is also called as **subscript** or **indices**.

We use the following general syntax to access individual elements of single dimensional array...

```
arrayName [ indexValue ]
```

### Example Code

```
marks [2] = 99 ;
```

In the above statement, the third element of '**marks**' array is assigned with value '**99**'.

## 9.3.2 MULTI DIMENSIONAL ARRAY

An array of arrays is called as multi dimensional array. In simple words, an array created with more than one dimension (size) is called as multi dimensional array. Multi dimensional array can be of **two dimensional array** or **three dimensional array** or **four dimensional array** or more...

Most popular and commonly used multi dimensional array is **two dimensional array**. The 2-D arrays are used to store data in the form of table. We also use 2-D arrays to create mathematical **matrices**.

### Declaration of Two Dimensional Array

We use the following general syntax for declaring a two dimensional array...

```
datatype arrayName [ rowSize ] [ columnSize ] ;
```

The above declaration of two dimensional array reserves 6 continuous memory locations of 2 bytes each in the form of **2 rows** and **3 columns**.

### Initialization of Two Dimensional Array

We use the following general syntax for declaring and initializing a two dimensional array with specific number of rows and columns with initial values.

```
datatype arrayName [rows][colms] = {{r1c1value, r1c2value,
..., {r2c1, r2c2, ...} ...} ;
```

## Notes

### Example Code

```
int matrix_A [2][3] = { {1, 2, 3},{4, 5, 6} } ;
```

The above declaration of two-dimensional array reserves 6 contiguous memory locations of 2 bytes each in the form of 2 rows and 3 columns. And the first row is initialized with values 1, 2 & 3 and second row is initialized with values 4, 5 & 6.

We can also initialize as follows...

### **Accessing Individual Elements of Two Dimensional Array**

In a C programming language, to access elements of a two-dimensional array we use array name followed by row index value and column index value of the element that to be accessed. Here the row and column index values must be enclosed in separate square braces. In case of the two-dimensional array the compiler assigns separate index values for rows and columns.

**arrayName [ rowIndex ] [ columnIndex ]**

We use the following general syntax to access the individual elements of a two-dimensional array...

### **Example Code**

```
matrix_A [0][1] = 10 ;
```

In the above statement, the element with row index 0 and column index 1 of **matrix\_A** array is assigned with value **10**.

### **Check your Progress-1**

1. Define Array

---

---

---



## 2. What is Multi Dimensional Array

---

---

---

---

## 9.4 APPLICATIONS OF ARRAYS IN C

---

In C programming language, arrays are used in wide range of applications. Few of them are as follows...

- **Arrays are used to Store List of values**

In c programming language, single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array data is stored in linear form.

- **Arrays are used to Perform Matrix Operations**

We use two dimensional arrays to create matrix. We can perform various operations on matrices using two dimensional arrays.

- **Arrays are used to implement Search Algorithms**

We use single dimensional arrays to implement search algorithms like ...

### 1. Linear Search

#### What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

#### Linear Search Algorithm (Sequential Search Algorithm)

Linear search algorithm finds a given element in a list of elements with  $O(n)$  time complexity where  $n$  is total number of elements in the list. This search process starts comparing search element with the first element in the list. If both are matched then result is element found

## Notes

otherwise search element is compared with the next element in the list. Repeat the same until search element is compared with the last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps...

- **Step 1** - Read the search element from the user.
- **Step 2** - Compare the search element with the first element in the list.
- **Step 3** - If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4** - If both are not matched, then compare search element with the next element in the list.
- **Step 5** - Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6** - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

### Example

Consider the following list of elements and the element to be searched...

list 

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

search element: 12

#### Step 1:

search element (12) is compared with first element (65)

list 

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

#### Step 2:

search element (12) is compared with next element (20)

list 

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

#### Step 3:

search element (12) is compared with next element (10)

list 

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

**Step 4:**

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7	
list	65	20	10	55	32	12	50	99	
				12					

Both are not matching. So move to next element

**Step 5:**

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7	
list	65	20	10	55	32	12	50	99	
					12				

Both are not matching. So move to next element

**Step 6:**

search element (12) is compared with next element (12)

## Implementation of Linear Search Algorithm using C Programming Language

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

void main(){
    int list[20],size,i,sElement;

    printf("Enter size of the list: ");
    scanf("%d",&size);

    printf("Enter any %d integer values: ",size);
    for(i = 0; i < size; i++)
        scanf("%d",&list[i]);

    printf("Enter the element to be Search: ");
    scanf("%d",&sElement);

    // Linear Search Logic
    for(i = 0; i < size; i++)
    {
        if(sElement == list[i])
        {
            printf("Element is found at %d index", i);
            break;
        }
    }
    if(i == size)
        printf("Given element is not found in the list!!!");
    getch();
}
```

### Output

```
C:\Users\User\Desktop\New folder\Search\bin\Debug\Search.exe
Enter size of the list: 5
Enter any 5 integer values: 3 1 5 7 4
Enter the element to be Search: 7
Element is found at 3 index
Process returned 0 (0x0) execution time : 77.741 s
Press any key to continue.
```

## 2. Binary Search

### Binary Search Algorithm

Binary search algorithm finds a given element in a list of elements with  $O(\log n)$  time complexity where  $n$  is total number of elements in the list. The binary search algorithm can be used with only a sorted list of elements. That means the binary search is used only with a list of elements that are already arranged in an order. The binary search can not be used for a list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for the left sublist of the middle element. If the search element is larger, then we repeat the same process for the right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1** - Read the search element from the user.
- **Step 2** - Find the middle element in the sorted list.
- **Step 3** - Compare the search element with the middle element in the sorted list.
- **Step 4** - If both are matched, then display "Given element is found!!!" and terminate the function.
- **Step 5** - If both are not matched, then check whether the search element is smaller or larger than the middle element.

- **Step 6** - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7** - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8** - Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9** - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

### Example

Consider the following list of elements and the element to be searched...

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

  
search element **12**

#### Step 1:

search element (12) is compared with middle element (50)

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

  
**12**

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

#### Step 2:

search element (12) is compared with middle element (12)

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

  
**12**

**Both are matching. So the result is "Element found at index 1"**

#### Step 1:

search element (80) is compared with middle element (50)

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

  
**80**

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

#### Step 2:

search element (80) is compared with middle element (65)

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

  
**80**

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

#### Step 3:

search element (80) is compared with middle element (80)

list 

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

  
**80**

**Both are not matching. So the result is "Element found at index 7"**

### Implementation of Binary Search Algorithm using C Programming Language

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

void main()
{
    int first, last, middle, size, i, sElement, list[100];
    clrscr();

    printf("Enter the size of the list: ");
    scanf("%d",&size);

    printf("Enter %d integer values in Assending order\n", s
size);

    for (i = 0; i < size; i++)
        scanf("%d",&list[i]);


    printf("Enter value to be search: ");
    scanf("%d", &sElement);

    first = 0;
    last = size - 1;
    middle = (first+last)/2;
```

```
while (first <= last) {
    if (list[middle] < sElement)
        first = middle + 1;
    else if (list[middle] == sElement) {
        printf("Element found at index %d.\n",middle);
        break;
    }
    else
        last = middle - 1;

    middle = (first + last)/2;
}
if (first > last)
    printf("Element Not found in the list.");
getch();
}
```

### Output



```
"C:\Users\User\Desktop\New folder\Search\bin\Debug\Search.exe"
Enter the size of the list: 5
Enter 5 integer values in Assending order
1 3 5 7 9
Enter value to be search: 3
Element found at index 1.
```

### Arrays are used to implement Sorting Algorithms

We use single dimensional arrays to implement sorting algorithms like ...

1. **Insertion Sort**
2. **Bubble Sort**

3. Selection Sort
4. Quick Sort
5. Merge Sort, etc.,

**Arrays are used to implement Datastructures**

We use single dimensional arrays to implement datastructures like...

1. Stack Using Arrays
2. Queue Using Arrays

**Arrays are also used to implement CPU Scheduling Algorithms**

**Check your Progress-1**

3. What is Search?

---

---

---

4. State 2 application of Arrays.

---

---

---

---

## **9.5 LET US SUM UP**

---

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. We have learnt different types of arrays and their applications.

---

## **9.6 KEYWORDS**

---

**Reserve** - retain for future use.

**Access** - obtain or retrieve (computer data or a file).

**Syntax** - the structure of statements in a computer language.

**Declaration** - a formal or explicit statement or announcement.

---

## 9.7 QUESTIONS FOR REVIEW

---

1. Explain Declaration of an Array
2. Explain Single Dimensional Array
3. What is Linear Search? Discuss in detail.

---

## 9.8 SUGGESTED READINGS AND REFERENCES

---

1. B. Gottfried: Programming with C , Tata McGraw-Hill Edition 2002.
2. E. Balagurusamy : Programming in ANSI C, Tata Mcgraw Hill - Edition 2002.
3. Brain W. Kernighan & Dennis M. Ritchie, The C Programme Language, 2nd Edition (ANSI features) , Prentice Hall 1989.
4. Let Us C- Y.P. Kanetkar, BPB Publication - 2002.
5. Analysis of Numerical Methods—Isacsons& Keller.
6. Numerical solutions of Ord. Diff. Equations—M K Jain
7. Numerical solutions of Partial Diff. Equations—G D Smith.
8. Programming with C, B. Gottfried, Tata-McGraw Hill
9. Programming with C, K. R. Venugopal and Sudeep R. Prasad, Tata-McGraw Hill

---

## 9.9 ANSWERS TO CHECK YOUR PROGRESS

---

1. Provide definition – 9.2
2. Provide explanation with example – 9.3.2
3. Provide definition – 9.4
4. Provide explanation – 9.4



---

# UNIT-10 POINTERS IN C

---

## STRUCTURE

10.0 Objectives

10.1 Introduction

10.2 Pointers in C

10.2.1 Declaring Pointers (Creating Pointers)

10.2.2 Assigning Address to Pointer

10.2.3 Accessing Variable Value Using Pointer

10.2.4 Memory Allocation of Pointer Variables

10.3 Pointers Arithmetic Operations in C

10.3.1 Addition Operation on Pointer

10.3.2 Subtraction Operation on Pointer

10.3.3 Increment & Decrement Operation on Pointer

10.3.4 Comparison of Pointers

10.4 Pointers to Pointers in C

10.5 Pointers to void in C

10.6 Pointers to Arrays in C

10.7 Pointers for Functions in C

10.8 Let us sum up

10.9 Keywords

10.10 Questions for Review

10.11 Suggested Reading and References

10.12 Answers to Check your Progress

---

## 10.0 OBJECTIVES

---

Understand the Pointers in C

## Notes

Comprehend the Pointers Arithmetic Operations in C

Understand the Pointers to void in C

Understand the application of pointers to array and function

---

### 10.1 INTRODUCTION

---

In the C programming language, we use normal variables to store user data values. When we declare a variable, the compiler allocates required memory with the specified name. In the c programming language, every variable has a name, datatype, value, storage class, and address. We use a special type of variable called a pointer to store the address of another variable with the same datatype.

---

### 10.2 POINTERS IN C

---

A pointer is defined as follows...

**Pointer is a special type of variable used to store the memory location address of a variable.**

In the C programming language, we can create pointer variables of any data type. Every pointer stores the address the variable with same data type only. That means integer pointer is used store the address of integer variable only.

#### **Accessing the Address of Variables**

In C programming language, we use the **reference operator "&"** to access the address of variable. For example, to access the address of a variable **"marks"** we use **"&marks"**. We use the following printf statement to display memory location address of variable **"marks"**...

#### **Example Code**

```
printf("Address : %u", &marks);
```

In the above example statement `%u` is used to display address of `marks` variable. Address of any memory location is unsigned integer value.

### 10.2.1 DECLARING POINTERS (CREATING POINTERS)

In C programming language, declaration of pointer variable is similar to the creation of normal variable but the name is prefixed with `*` symbol. We use the following syntax to declare a pointer variable...

```
datatype *pointerName ;
```

A variable declaration prefixed with `*` symbol becomes a pointer variable.

#### Example Code

```
int *ptr ;
```

In the above example declaration, the variable `"ptr"` is a pointer variable that can be used to store any integer variable address.

### 10.2.2 ASSIGNING ADDRESS TO POINTER

To assign address to a pointer variable we use assignment operator with the following syntax...

```
pointerVariableName = & variableName ;
```

For example, consider the following variables declaration...

Example Program | Test whether given number is divisible by 5.

```
int a, *ptr ;
```

In the above declaration, variable `"a"` is a normal integer variable and variable `"ptr"` is an integer pointer variable. If we want to assign the address of variable `"a"` to pointer variable `"ptr"` we use the following statement...

#### Example Code

## Notes

```
ptr = &a ;
```

In the above statement, the address of variable "**a**" is assigned to pointer variable "**ptr**". Here we say that pointer variable **ptr** is pointing to variable **a**.

### 10.2.3 ACCESSING VARIABLE VALUE USING POINTER

Pointer variables are used to store the address of other variables. We can use this address to access the value of the variable through its pointer. We use the symbol "\*" in front of pointer variable name to access the value of variable to which the pointer is pointing. We use the following general syntax...

**\*pointerVariableName**

#### Example Code

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

void main()
{
    int a = 10, *ptr ;
    clrscr();
    ptr = &a ;

    printf("Address of variable a = %u\n", ptr) ;
    printf("Value of variable a = %d\n", *ptr) ;
    printf("Address of variable ptr = %u\n", &ptr) ;
}
```

#### Output



```
"C:\Users\User\Desktop\New folder\pointer_example\bin\Debug\pointer_example.exe"
Address of variable a = 6356748
Value of variable a = 10
Address of variable ptr = 6356744

Process returned 0 (0x0)   execution time : 0.087 s
Press any key to continue.
```

In the above example program, variable **a** is a normal variable and variable **ptr** is a pointer variable. Address of variable **a** is stored in

pointer variable **ptr**. Here **ptr** is used to access the address of variable **a** and **\*ptr** is used to access the value of variable **a**.

#### 10.2.4 MEMORY ALLOCATION OF POINTER VARIABLES

Every pointer variable is used to store the address of another variable. In computer memory address of any memory location is an **unsigned integer** value. In C programming language, unsigned integer requires **2 bytes** of memory. So, irrespective of pointer datatype every pointer variable is allocated with 2 bytes of memory.

---

### 10.3 POINTERS ARITHMETIC OPERATIONS IN C

---

Pointer variables are used to store the address of variables. Address of any variable is an unsigned integer value i.e., it is a numerical value. So we can perform arithmetic operations on pointer values. But when we perform arithmetic operations on pointer variable, the result depends on the amount of memory required by the variable to which the pointer is pointing.

In the C programming language, we can perform the following arithmetic operations on pointers...

1. Addition
2. Subtraction
3. Increment
4. Decrement
5. Comparison

#### 10.3.1 ADDITION OPERATION ON POINTER

In the C programming language, the addition operation on pointer variables is calculated using the following formula...

$$\text{AddressAtPointer} + (\text{NumberToBeAdd} * \text{BytesOfMemoryRequiredByDatatype})$$

### Example Program

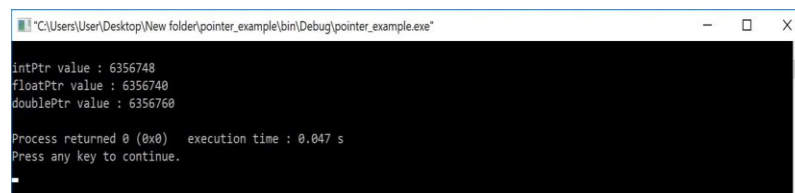
```
void main()
{
    int a, *intPtr ;
    float b, *floatPtr ;
    double c, *doublePtr ;
    clrscr() ;
    intPtr = &a ; // Asume address of a is 1000
    floatPtr = &b ; // Asume address of b is 2000
    doublePtr = &c ; // Asume address of c is 3000

    intPtr = intPtr + 3 ; // intPtr = 1000 + ( 3 * 2 )
    floatPtr = floatPtr + 2 ; // floatPtr = 2000 + ( 2 *
4 )
    doublePtr = doublePtr + 5 ; // doublePtr = 3000 + (
5 * 6 )

    printf("intPtr value : %u\n", intPtr) ;
    printf("floatPtr value : %u\n", floatPtr) ;
    printf("doublePtr value : %u", doublePtr) ;

    getch() ;
}
```

### Output:



```
"C:\Users\User\Desktop\New folder\pointer_example\bin\Debug\pointer_example.exe"
intPtr value : 6356748
floatPtr value : 6356740
doublePtr value : 6356760
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

### 10.3.2 SUBTRACTION OPERATION ON POINTER

In the C programming language, the subtraction operation on pointer variables is calculated using the following formula...

$$\text{AddressAtPointer} - (\text{NumberToBeAdd} * \text{BytesOfMemoryRequiredByDatatype})$$

### Example Program

```

void main()
{
    int a, *intPtr ;
    float b, *floatPtr ;
    double c, *doublePtr ;
    clrscr() ;
    intPtr = &a ; // Asume address of a is 1000
    floatPtr = &b ; // Asume address of b is 2000
    doublePtr = &c ; // Asume address of c is 3000

    intPtr = intPtr - 3 ; // intPtr = 1000 - ( 3 * 2 )
    floatPtr = floatPtr - 2 ; // floatPtr = 2000 - ( 2 *
4 )
    doublePtr = doublePtr - 5 ; // doublePtr = 3000 - (
5 * 6 )

    printf("intPtr value : %u\n", intPtr) ;
    printf("floatPtr value : %u\n", floatPtr) ;
    printf("doublePtr value : %u", doublePtr) ;

    getch() ;
}

```

### Output

```

"C:\Users\User\Desktop\New folder\pointer_example\bin\Debug\pointer_example.exe"
intPtr value : 6356724
floatPtr value : 6356724
doublePtr value : 6356680

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.

```

### 10.3.3 INCREMENT & DECREMENT OPERATION ON POINTER

The increment operation on pointer variable is calculated as follows...

AddressAtPointer +  
NumberOfBytesRequiresByDatatype

### Example Program

```

void main()
{
    int a, *intPtr ;
    float b, *floatPtr ;
    double c, *doublePtr ;
    clrscr() ;
    intPtr = &a ; // Asume address of a is 1000
    floatPtr = &b ; // Asume address of b is 2000
    doublePtr = &c ; // Asume address of c is 3000

    intPtr++; // intPtr = 1000 + 2
    floatPtr++; // floatPtr = 2000 + 4
    doublePtr++; // doublePtr = 3000 + 6

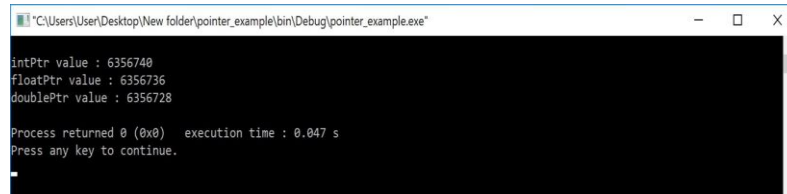
    printf("intPtr value : %u\n", intPtr) ;
    printf("floatPtr value : %u\n", floatPtr) ;
    printf("doublePtr value : %u", doublePtr) ;

    getch() ;
}

```

## Notes

### Output



```
"C:\Users\User\Desktop\New folder\pointer_example\bin\Debug\pointer_example.exe"
intPtr value : 6356740
floatPtr value : 6356736
doublePtr value : 6356728

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

The decrement operation on pointer variable is calculated as follows...

### AddressAtPointer - NumberOfBytesRequiresByDatatype

#### Example Program

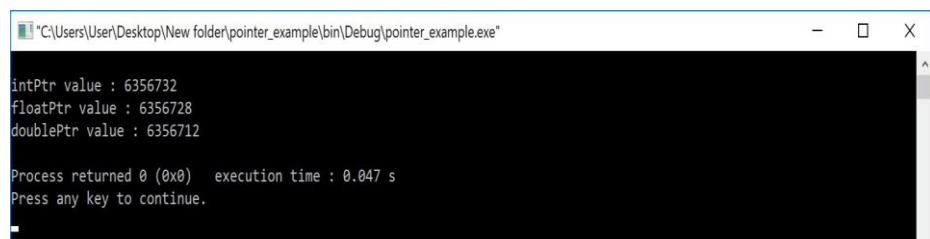
```
void main()
{
    int a, *intPtr ;
    float b, *floatPtr ;
    double c, *doublePtr ;
    clrscr() ;
    intPtr = &a ; // Asume address of a is 1000
    floatPtr = &b ; // Asume address of b is 2000
    doublePtr = &c ; // Asume address of c is 3000

    intPtr-- ; // intPtr = 1000 - 2
    floatPtr-- ; // floatPtr = 2000 - 4
    doublePtr-- ; // doublePtr = 3000 - 6

    printf("intPtr value : %u\n", intPtr) ;
    printf("floatPtr value : %u\n", floatPtr) ;
    printf("doublePtr value : %u", doublePtr) ;

    getch() ;
}
```

### Output



```
"C:\Users\User\Desktop\New folder\pointer_example\bin\Debug\pointer_example.exe"
intPtr value : 6356732
floatPtr value : 6356728
doublePtr value : 6356712

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

## 10.3.4 COMPARISON OF POINTERS



The comparison operation is performed between the pointers of same datatype only. In C programming language, we can use all comparison operators (relational operators) with pointers.

[NOTE: We can't perform multiplication and division operations on pointers.]

### Check your Progress-1

1. Explain Pointer.

---

---

---

2. Discuss addition operation on pointer

---

---

---

---

## 10.4 POINTERS TO POINTERS IN C

---

In the C programming language, we have pointers to store the address of variables of any datatype. A pointer variable can store the address of a normal variable. C programming language also provides a pointer variable to store the address of another pointer variable. This type of pointer variable is called a pointer to pointer variable. Sometimes we also call it a double pointer. We use the following syntax for creating pointer to pointer...

```
datatype **pointerName ;
```

### Example Program

```
int **ptr ;
```

Here, **ptr** is an integer pointer variable that stores the address of another integer pointer variable but does not store the normal integer variable address.

## Notes

### **MOST IMPORTANT POINTS TO BE REMEMBERED**

1. To store the address of normal variable we use single pointer variable
2. To store the address of single pointer variable we use double pointer variable
3. To store the address of double pointer variable we use triple pointer variable
4. Similarly the same for remaining pointer variables also...

### Example Program


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

int main()
{
    int a ;
    int *ptr1 ;
    int **ptr2 ;
    int ***ptr3 ;

    ptr1 = &a ;
    ptr2 = &ptr1 ;
    ptr3 = &ptr2 ;

    printf("\nAddress of normal variable 'a' = %u\n", ptr1) ;
    printf("Address of pointer variable '*ptr1' = %u\n", ptr2) ;
    printf("Address of pointer-to-pointer '**ptr2' = %u\n", ptr3) ;
    return 0;
}
```

### Output:



```
C:\Users\User\Desktop\New folder\pointer_pointer\bin\Debug\pointer_pointer.exe
Address of normal variable 'a' = 6356744
Address of pointer variable '*ptr1' = 6356748
Address of pointer-to-pointer '**ptr2' = 6356736
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

---

## 10.5 POINTERS TO VOID IN C

---

In the C programming language, pointer to void is the concept of defining a pointer variable that is independent of data type. In C programming language, a void pointer is a pointer variable used to store the address of a variable of any datatype. That means single void pointer can be used to store the address of integer variable, float variable,

character variable, double variable or any structure variable. We use the keyword **"void"** to create void pointer. We use the following syntax for creating a pointer to void...

```
void *pointerName ;
```

### Example Code

```
void *ptr ;
```

Here, **"ptr"** is a void pointer variable which is used to store the address of any datatype variable.

### MOST IMPORTANT POINTS TO BE REMEMBERED

1. void pointer stores the address of any datatype variable.

#### Example Program

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

int main()
{
    int a ;
    float b ;
    char c ;

    void *ptr ;

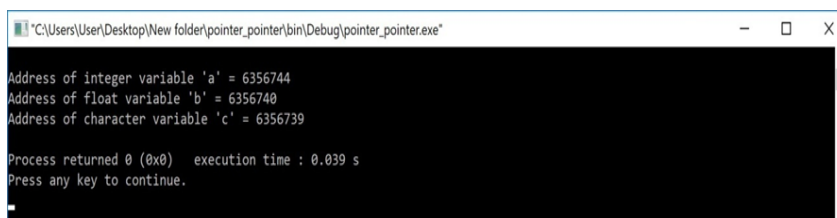
    clrscr() ;

    ptr = &a ;
    printf("Address of integer variable 'a' = %u\n", ptr)
;

    ptr = &b ;
    printf("Address of float variable 'b' = %u\n", ptr) ;

    ptr = &c ;
    printf("Address of character variable 'c' = %u\n", ptr) ;
    return 0;
}
```

Output:



```
C:\Users\User\Desktop\New folder\pointer_pointer\bin\Debug\pointer_pointer.exe
Address of integer variable 'a' = 6356744
Address of float variable 'b' = 6356740
Address of character variable 'c' = 6356739
Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

---

## 10.6 POINTERS TO ARRAYS IN C

---

In the C programming language, when we declare an array the compiler allocates the required amount of memory and also creates a constant pointer with array name and stores the base address of that pointer in it. The address of the first element of an array is called as **base address** of that array.

The array name itself acts as a pointer to the first element of that array. Consider the following example of array declaration...

### Example Code

```
int marks[6] ;
```

For the above declaration, the compiler allocates 12 bytes of memory and the address of first memory location (i.e., marks[0]) is stored in a constant pointer called **marks**. That means in the above example, **marks** is a pointer to **marks[0]**.

### Example Program

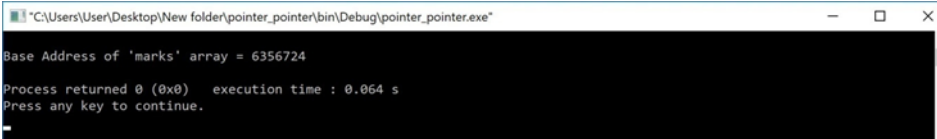
```
#include<stdio.h>
#include<conio.h>

int main()
{
    int marks[6] = {89, 45, 58, 72, 90, 93} ;
    int *ptr ;

    clrscr() ;

    ptr = marks ;
    printf("Base Address of 'marks' array = %u\n", ptr) ;
    return 0;
}
```

### Output:



```
"C:\Users\User\Desktop\New folder\pointer_pointer\bin\Debug\pointer_pointer.exe"
Base Address of 'marks' array = 6356724
Process returned 0 (0x0) execution time : 0.064 s
Press any key to continue.
```

**MOST IMPORTANT POINTS TO BE REMEMBERED**

1. An array name is a **constant pointer**.
2. We can use the array name to access the address and value of all the elements of that array.
3. Since array name is a constant pointer we can't modify its value.

Consider the following example statements...

**Example Code**

```
ptr = marks + 2 ;
```

Here, the pointer variable "**ptr**" is assigned with address of "**marks[2]**" element

**Example Code**

```
printf("Address of 'marks[4]' = %u", marks+4) ;
```

The above printf statement displays the address of element "**marks[4]**".

**EXAMPLE CODE**

```
printf("Value of 'marks[0]' = %d", *marks) ;
```

```
printf("Value of 'marks[3]' = %d", *(marks+3)) ;
```

In the above two statements, first printf statement prints the value **89** (i.e., value of marks[0]) and the second printf statement prints the value **72** (i.e., value of marks[3]).

**EXAMPLE CODE**

```
marks++ ;
```

The above statement generates **compilation error** because the array name acts as a constant pointer. So we can't change its value.

## Notes

In the above example program, the array name **marks** can be used as follows...

<b>marks</b> is		same		as <b>&amp;marks[0]</b>
<b>marks</b>	+	<b>1</b> is	same	as <b>&amp;marks[1]</b>
<b>marks</b>	+	<b>2</b> is	same	as <b>&amp;marks[2]</b>
<b>marks</b>	+	<b>3</b> is	same	as <b>&amp;marks[3]</b>
<b>marks</b>	+	<b>4</b> is	same	as <b>&amp;marks[4]</b>
<b>marks</b>	+	<b>5</b> is	same	as <b>&amp;marks[5]</b>
<b>*marks</b> is		same		as <b>marks[0]</b>
<b>*(marks</b>	+	<b>1)</b> is	same	as <b>marks[1]</b>
<b>*(marks</b>	+	<b>2)</b> is	same	as <b>marks[2]</b>
<b>*(marks</b>	+	<b>3)</b> is	same	as <b>marks[3]</b>
<b>*(marks</b>	+	<b>4)</b> is	same	as <b>marks[4]</b>
<b>*(marks + 5)</b> is same as <b>marks[5]</b>				

### 10.6.1 POINTERS TO MULTI DIMENSIONAL ARRAY

In case of multi-dimensional array also the array name acts as a constant pointer to the base address of that array. For example, we declare an array as follows...

#### EXAMPLE CODE

```
int marks[3][3];
```

In the above example declaration, the array name **marks** acts as constant pointer to the base address (**address of marks[0][0]**) of that array. In the above example of two dimensional array, the element **marks[1][2]** is accessed as **\*(\*(marks + 1) + 2)**.

---

## 10.7 POINTERS FOR FUNCTIONS IN C

---

In the C programming language, there are two ways to pass parameters to functions. They are as follows...

1. Call by Value

## 2. Call By Reference

We use pointer variables as formal parameters in **call by reference** parameter passing method.

In case of **call by reference** parameter passing method, the address of actual parameters is passed as arguments from the calling function to the called function. To receive this address, we use pointer variables as formal parameters.

Consider the following program for swapping two variable values...

### Example - Swapping of two variable values using Call by Reference

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

void swap(int *, int *) ;

void main()
{
    int a = 10, b = 20 ;

    clrscr() ;

    printf("Before swap : a = %d and b = %d\n", a, b) ;

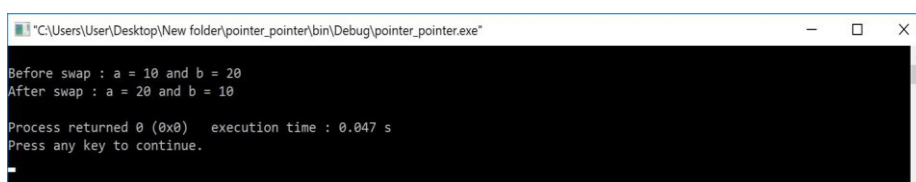
    swap(&a, &b) ;

    printf("After swap : a = %d and b = %d\n", a, b) ;

    getch() ;
}

void swap(int *x, int *y)
{
    int temp ;
    temp = *x ;
    *x = *y ;
    *y = temp ;
}
```

### Output:



```
C:\Users\User\Desktop\New folder\pointer_pointer\bin\Debug\pointer_pointer.exe
Before swap : a = 10 and b = 20
After swap : a = 20 and b = 10
Process returned 0 (0x0) execution time : 0.047 s
Press any key to continue.
```

In the above example program, we are passing the addresses of variables **a** and **b** and these are received by the pointer variables **x** and **y**.

## Notes

In the called function **swap** we use the pointer variables **x** and **y** to swap the values of variables **a** and **b**.

### Check your Progress-1

3. What is Double Pointer.

---

---

---

4. Define base address& marks

---

---

---

---

## 10.8 LET US SUM UP

---

We learnt about pointers- how to create, access and address the pointers. Enumerate its various application in C programming. Application of pointers in Arrays and functions.

---

## 10.9 KEYWORDS

---

**Create** - bring (something) into existence.

**Argument** - the term **argument** refers to any expression within the parentheses of a function call

**Swap** - the act of **swapping** two variables refers to mutually exchanging the values of the variables.

---

## 10.10 QUESTIONS FOR REVIEW

---

1. Explain accessing variable value using pointer
2. Discuss increment & decrement operation on pointer



3. What do you understand by 'pointers to void in C'?
4. Explain Pointers for Functions in C

---

## 10.11 SUGGESTED READINGS AND REFERENCES

---

1. B. Gottfried: Programming with C , Tata McGraw-Hill Edition 2002.
2. E. Balagurusamy : Programming in ANSI C, Tata Mcgraw Hill - Edition 2002.
3. Brain W. Kernighan & Dennis M. Ritchie, The C Programme Language, 2nd Edition (ANSI features) , Prentice Hall 1989.
4. Let Us C- Y.P. Kanetkar, BPB Publication - 2002.
5. Analysis of Numerical Methods—Isacsons& Keller.
6. Numerical solutions of Ord. Diff. Equations—M K Jain
7. Numerical solutions of Partial Diff. Equations—G D Smith.
8. Programming with C, B. Gottfried, Tata-McGraw Hill
9. Programming with C, K. R. Venugopal and Sudeep R. Prasad, Tata-McGraw Hill

---

## 10.12 ANSWERS TO CHECK YOUR PROGRESS

---

1. Provide explanation and definition – 10.2
2. Provide explanation with example – 10.3.1
3. Provide explanation – 10.4
4. Provide definition – 10.6

---

# UNIT-11 DYNAMIC MEMORY ALLOCATION AND STRINGS IN C

---

## STRUCTURE

11.0 Objectives

11.1 Introduction

11.2 Dynamic Memory Allocation in C

11.2.1 malloc()

11.2.2 calloc()

11.2.3 realloc()

11.2.4 free()

11.3 String in C

11.3.1 Creating string in C programming language

11.3.2 Assigning string value in c programming language

11.3.3 Reading string value from user in c programming language

11.4 String Handling Functions in C

11.5 Enumerated Types (enum) in C

11.6 typedef in C

11.6.1 typedef with primitive datatypes

11.6.2 typedef int Number

11.6.3 typedef with Arrays

11.6.4 typedef with user defined datatypes like structures, unions  
etc.,

11.6.5 typedef with Pointers

11.7 Application of Arrays in C

11.8 Let us sum up

11.9 Keywords

11.10 Questions for Review

11.11 Suggested Reading and References

11.12 Answers to Check your Progress

---

## 11.0 OBJECTIVES

---

Understand the Dynamic Memory Allocation in C

Understand the concept of String and String Handling functions in C

Comprehend Enumeration, typedef and application of arrays

---

## 11.1 INTRODUCTION

---

In C programming language, when we declare variables memory is allocated in space called stack. The memory allocated in the stack is fixed at the time of compilation and remains until the end of the program execution. When we create an array, we must specify the size at the time of the declaration itself and it cannot be changed during the program execution. This is a major problem when we do not know the number of values to be stored in an array. To solve this we use the concept of **Dynamic Memory Allocation**

---

## 11.2 DYNAMIC MEMORY ALLOCATION IN C

---

The dynamic memory allocation allocates memory from **heap storage**. Dynamic memory allocation is defined as follow...

**Allocation of memory during the program execution is called dynamic memory allocation.**

Or

**Dynamic memory allocation is the process of allocating the memory manually at the time of program execution.**

## Notes

We use pre-defined or standard library functions to allocate memory dynamically. There are **FOUR** standard library functions that are defined in the header file known as "**stdlib.h**". They are as follows...

```
malloc()
calloc()
realloc()
free()
```

### 11.2.1 malloc()

malloc() is the standard library function used to allocate a memory block of specified number of bytes and returns void pointer. The void pointer can be casted to any datatype. If malloc() function unable to allocate memory due to any reason it returns NULL pointer.

#### Syntax

```
void* malloc(size_in_bytes)
```

#### Example Program for *malloc()*.

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

int main () {

    char *title;

    title = (char *) malloc(15);

    strcpy(title, "c programming");
    printf("String = %s, Address = %u\n", title, title);

    return(0);

}
```

### 11.2.2 calloc()

calloc() is the standard library function used to allocate multiple memory blocks of the specified number of bytes and initializes them to ZERO.

calloc() function returns void pointer. If calloc() function unable to allocate memory due to any reason it returns a NULL pointer. Generally, calloc() is used to allocate memory for array and structure. calloc() function takes two arguments and they are 1. The number of blocks to be allocated, 2. Size of each block in bytes

### Syntax

```
void* calloc(number_of_blocks,
             size_of_each_block_in_bytes)
```

#### Example Program for *calloc()*.

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

int main () {
    int i, n;
    int *ptr;

    printf("Number of blocks to be created:");
    scanf("%d",&n);

    ptr = (int*)calloc(n, sizeof(int));
    printf("Enter %d numbers:\n",n);
    for( i=0 ; i < n ; i++ ) {
        scanf("%d",&ptr[i]);
    }

    printf("The numbers entered are: ");
    for( i=0 ; i < n ; i++ ) {
        printf("%d ",ptr[i]);
    }

    return(0);
}
```

### 11.2.3 realloc()

realloc() is the standard library function used to modify the size of memory blocks that were previously allocated using malloc() or calloc(). realloc() function returns void pointer. If calloc() function unable to allocate memory due to any reason it returns NULL pointer.

### Syntax

```
void* realloc(*pointer,
             new_size_of_each_block_in_bytes)
```

### Example Program for *realloc()*.

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

int main () {

    char *title;

    title = (char *) malloc(15);

    strcpy(title, "c programming");
    printf("Before modification : String = %s, Address =
    %u\n", title, title);

    title = (char*) realloc(title, 30);

    strcpy(title,"C Programming Language");
    printf("After modification : String = %s, Address =
    %u\n", title, title);

    return(0);

}
```

### 11.2.4 free()

free() is the standard library function used to deallocate memory block that was previously allocated using malloc() or calloc(). free() function returns void pointer. When free() function is used with memory allocated that was created using calloc(), all the blocks are get deallocated.

#### Syntax

void free(\*pointer)

### Example Program for *free()*.

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

int main () {

    char *title;

    title = (char *) malloc(15);

    strcpy(title, "c programming");
    printf("Before modification : String = %s, Address =
    %u\n", title, title);

    title = (char*) realloc(title, 30);

    strcpy(title,"C Programming Language");
    printf("After modification : String = %s, Address =
    %u\n", title, title);

    free(title);

    return(0);

}
```

**Check your Progress-1**

1. State Dynamic memory allocation in C

---

---

---

2. Explain calloc()

---

---

---

---

## 11.3 STRINGS IN C

---

String is a set of characters that are enclosed in double quotes. In the C programming language, strings are created using one dimension array of character datatype. Every string in C programming language is enclosed within double quotes and it is terminated with NULL (\0) character. Whenever c compiler encounters a string value it automatically appends a NULL character (\0) at the end. The formal definition of string is as follows...

**String is a set of characters enclosed in double quotation marks. In C programming, the string is a character array of single dimension.**

In C programming language, there are two methods to create strings and they are as follows...

1. Using one dimensional array of character datatype ( static memory allocation )
2. Using a pointer array of character datatype ( dynamic memory allocation )

### 11.3.1 CREATING STRING IN C PROGRAMMING LANGUAGE

In C, strings are created as a one-dimensional array of character datatype. We can use both static and dynamic memory allocation. When we create

```
char str[6];
```

a string, the size of the array must be one more than the actual number of characters to be stored. That extra memory block is used to store string termination character NULL (\0). The following declaration stores a string of size 5 characters.

The following declaration creates a string variable of a specific size at the time of program execution.

```
char *str = (char *) malloc(15);
```

### 11.3.2 ASSIGNING STRING VALUE IN C PROGRAMMING LANGUAGE

String value is assigned using the following two methods...

1. At the time of declaration (initialization)
2. After declaration

#### Examples of assigning string value

```
int main()
{
    char str1[6] = "Hello";
    char str2[] = "Hello!";
    char name1[] = {'s','m','a','r','t'};
    char name2[6] = {'s','m','a','r','t'};

    char title[20];
    *title = "btech smart class";

    return 0;
}
```

### 11.3.3 READING STRING VALUE FROM USER IN C PROGRAMMING LANGUAGE

We can read a string value from the user during the program execution.

We use the following two methods...

1. Using **scanf() method - reads single word**



## 2. Using **gets()** method - reads a line of text

Using `scanf()` method we can read only one word of string. We use `%s` to represent string in `scanf()` and `printf()` methods.

### Examples of reading string value using `scanf()` method

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

int main(){

    char name[50];
    printf("Please enter your name : ");

    scanf("%s", name);

    printf("Hello! %s , welcome to btech smart class !!",
name);

    return 0;
}
```

When we want to read multiple words or a line of text, we use a pre-defined method `gets()`. The `gets()` method terminates the reading of text with **Enter** character.

### Examples of reading string value using `gets()` method

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

int main(){

    char name[50];
    printf("Please enter your name : ");

    gets(name);

    printf("Hello! %s , welcome to btech smart class !!",
name);

    return 0;
}
```

C Programming language provides a set of pre-defined functions called **String Handling Functions** to work with string values. All the string handling functions are defined in a header file called **string.h**.

---

## 11.4 STRING HANDLING FUNCTIONS IN C

---

C programming language provides a set of pre-defined functions called **string handling functions** to work with string values. The string handling functions are defined in a header file called **string.h**. Whenever we want to use any string handling function we must include the header file called **string.h**.

The following table provides most commonly used string handling function and their use...

Function	Syntax (or) Example	Description
strcpy()	strcpy(string1, string2)	Copies string2 value into string1
strncpy()	strncpy(string1, string2, 5)	Copies first 5 characters string2 into string1
strlen()	strlen(string1)	returns total number of characters in string1
strcat()	strcat(string1,string2)	Appends string2 to string1
strncat()	strncpy(string1, string2, 4)	Appends first 4 characters of string2 to string1
strcmp()	strcmp(string1, string2)	Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2
strncmp()	strncmp(string1, string2, 4)	Compares first 4 characters of both string1 and string2
strncmpi()	strncmpi(string1,string2)	Compares two strings, string1 and string2 by ignoring case (upper or

		lower)
stricmp()	stricmp(string1, string2)	Compares two strings, string1 and string2 by ignoring case (similar to strcmpi())
strlwr()	strlwr(string1)	Converts all the characters of string1 to lower case.
strupr()	strupr(string1)	Converts all the characters of string1 to upper case.
strdup()	string1 = strdup(string2)	Duplicated value of string2 is assigned to string1
strchr()	strchr(string1, 'b')	Returns a pointer to the first occurrence of character 'b' in string1
strrchr()	'strrchr(string1, 'b')	Returns a pointer to the last occurrence of character 'b' in string1
strstr()	strstr(string1, string2)	Returns a pointer to the first occurrence of string2 in string1
strset()	strset(string1, 'B')	Sets all the characters of string1 to given character 'B'.
strnset()	strnset(string1, 'B', 5)	Sets first 5 characters of string1 to given character 'B'.
strrev()	strrev(string1)	It reverses the value of string1

---

## 11.5 ENUMERATED TYPES (ENUM) IN C

---

In C programming language, an enumeration is used to create user-defined datatypes. Using enumeration, integral constants are assigned with names and we use these names in the program. Using names in programming makes it more readable and easy to maintain.

## Notes

**Enumeration is the process of creating user defined datatype by assigning names to integral constants**

We use the keyword **enum** to create enumerated datatype. The general syntax of enum is as follows...

```
enum {name1, name2, name3, ... }
```

In the above syntax, integral constant '0' is assigned to name1, integral constant '1' is assigned to name2 and so on. We can also assign our own integral constants as follows...

```
enum {name1 = 10, name2 = 30, name3 = 15, ... }
```

In the above syntax, integral constant '10' is assigned to name1, integral constant '30' is assigned to name2 and so on.

### Example Program for enum with default values

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

enum day { Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday} ;

void main(){

    enum day today;

    today = tuesday ;

    printf("\ntoday = %d ", today) ;

}
```

In the above example program a user defined datatype "**day**" is created with seven values, Monday as integral constant '0', Tuesday as integral constant '1', Wednesday as integral constant '2', Thursday as integral constant '3',

friday as integral constant '4', saturday as integral constant '5' and sunday as integral constant '6'. Here, when we display tuesday it displays the respective integral constant '1'.

We can also change the order of integral constants, consider the following example program.

### Example Program for enum with changed integral constant values

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

enum day { Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} ;

void main(){

    enum day today;

    today = tuesday ;

    printf("\ntoday = %d ", today) ;

}
```

In the above example program, the integral constant value starts with '1' instead of '0'. Here, tuesday value is displayed as '2'.

We can also create enum with our own integral constants, consider the following example program.

### Example Program for enum with defferent integral constant values

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

enum threshold {low = 40, normal = 60, high = 100} ;

void main(){

    enum threshold status;

    status = low ;

    printf("\nYou are at %d state. Work hard to improve!!", status) ;

}
```

## Notes

Some times we may assign our own integral constant from other than the first name. In this case, compiler follows default integral constants for the name that is before the user-defined integral constant and from user-defined constant onwards it resumes. Consider the following example program.

**Example Program for enum with changed integral constant values**

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

enum day {Monday, Tuesday, Wednesday, Thursday = 10, Friday, Saturday, Sunday} ;

void main(){

    enum day today;

    today = tuesday ;

    printf("\ntoday = %d ", today) ;

    today = saturday ;

    printf("\ntoday = %d ", today) ;

}
```

In the above example program a user defined datatype "**day**" is created seven values, monday as integral constant '0', tuesday as integral constant '1', wednesday as integral constant '2', thursday as integral constant '10', friday as integral constant '11', saturday as integral constant '12' and sunday as integral constant '13'.

Note - In enumeration, more than one name may give with same integral constant.

---

## 11.6 TYPEDEF IN C

---

In C programming language, **typedef** is a keyword used to create alias name for the existing datatypes. Using **typedef** keyword we can create a temporary name to the system defined datatypes like int, float, char and double. we use that temporary name to create a variable. The general syntax of typedef is as follows...

**typedef <existing-datatype> <alias-name>**

### 11.6.1 typedef with primitive datatypes

Consider the following example of typedef

#### Example Program to illustrate typedef in C.

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

typedef int Number;

void main(){

    Number a,b,c;    // Here a,b,&c are integer type of v
    ariables.

    clrscr() ;
    printf("Enter any two integer numbers: ") ;
    scanf("%d%d", &a,&b) ;

    c = a + b;

    printf("Sum = %d", c) ;
}
```

### 11.6.2 typedef int Number

In the above example, **Number** is defined as alias name for integer datatype. So, we can use **Number** to declare integer variables.

#### Example Program to illustrate typedef with arrays in C.

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

void main(){

    typedef int Array[5];    // Here Array acts like an i
    nteger array type of size 5.

    Array list = {10,20,30,40,50};    // List is an array
    of integer type with size 5.
    int i;

    clrscr() ;

    printf("List elements are : \n") ;

    for(i=0; i<5; i++)
        printf("%d\t", list[i]) ;
}
```

## Notes

In the above example program, **Array** is the alias name of integer array type of size 5. We can use **Array** as datatype to create integer array of size 5. Here, list is an integer array of size 5.

### 11.6.3 typedef with Arrays

In C programming language, typedef is also used with arrays. Consider the following example program to understand how typedef is used with arrays.

#### Example Program to illustrate typedef with arrays in C.

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

typedef struct student
{
    char stud_name[50];
    int stud_rollNo;
}stud;

void main(){

    stud s1;
    clrscr() ;

    printf("Enter the student name: ") ;
    scanf("%s", s1.stud_name);
    printf("Enter the student Roll Number: ");
    scanf("%d", &s1.stud_rollNo);

    printf("\nStudent Information\n");
    printf("Name - %s\nHallticket Number - %d", s1.stud_n
ame, s1.stud_rollNo);
}
```

### 11.6.4 typedef with user defined datatypes like structures, unions etc.,

In C programming language, typedef is also used with structures and unions. Consider the following example program to understand how typedef is used with structures.



In the above example program, **stud** is the alias name of student structure. We can use **stud** as datatype to create variables of student structure. Here, **s1** is a variable of student structure datatype.

### 11.6.5 typedef with Pointers

In C programming language, typedef is also used with pointers. Consider the following example program to understand how typedef is used with pointers.

#### Example Program to illustrate typedef with Pointers in C.

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

void main(){

    typedef int* intPointer;

    intPointer ptr; // Here ptr is a pointer of integer
    datatype.
    int a = 10;

    clrscr() ;

    ptr = &a;

    printf("Address of a = %u ",ptr) ;
    printf("\nValue of a = %d ",*ptr);
}
```

#### Check your Progress-2

3. What is String? State its methods to create the strings.

---

---

---

4. Give description of strcmp()

---

---

---

5. Explain Enumeration

---

---

---

---

### 11.7 LET US SUM UP

---

We explored dynamic memory allocation concept in C. In the C programming language, strings are created using one dimension array of character datatype. We came across different string handling functions. Using enumeration, integral constants are assigned with names. We learnt about **typedef** keyword we can create a temporary name to the system defined datatypes.

---

### 11.8 KEYWORDS

---

**Predefined** functions **means** functions which we will not **define** but we can use those functions in our code by including some header files because they have a **definition** in header files called (filename. h)

**Enclosed** - to close in

**Alias** - a false or assumed identity.

**Data type** - a particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

---

### 11.9 QUESTIONS FOR REVIEW

---

1. Explain realloc() & free() standard library function
2. State two methods to create strings
3. Brief about **string handling functions**
4. Explain Pointers for Functions in C

5. Explain typedef in C

---

## 11.10 SUGGESTED READINGS AND REFERENCES

---

1. B. Gottfried: Programming with C , Tata McGraw-Hill Edition 2002.
2. E. Balagurusamy : Programming in ANSI C, Tata Mcgraw Hill - Edition 2002.
3. Brain W. Kernighan & Dennis M. Ritchie, The C Programme Language, 2nd Edition (ANSI features) , Prentice Hall 1989.
4. Let Us C- Y.P. Kanetkar, BPB Publication - 2002.
5. Analysis of Numerical Methods—Isacsons& Keller.
6. Numerical solutions of Ord. Diff. Equations—M K Jain
7. Numerical solutions of Partial Diff. Equations—G D Smith.
8. Programming with C, B. Gottfried, Tata-McGraw Hill
9. Programming with C, K. R. Venugopal and Sudeep R. Prasad, Tata-McGraw Hill

---

## 11.11 ANSWERS TO CHECK YOUR PROGRESS

---

1. Provide definition – 11.2
2. Provide explanation with example – 11.2.2
3. Provide explanation – 11.3
4. Provide explanation in the table – 11.4
5. Provide definition – 11.5

---

# UNIT-12 STRUCTURE IN C AND FILE HANDLING

---

## STRUCTURE

12.0 Objectives

12.1 Introduction

12.2 Structure in C

12.2.1 How to create structure?

12.2.2 Creating and Using structure variables

12.2.3 Memory allocation of structure

12.3 Union in C

12.3.1 How to create union?

12.3.2 Creating and Using union variables

12.3.3 Memory allocation of union

12.4 Bit Field in C

12.5 Command Line Argument in C

12.6 Files in C

12.7 File Handling Functions in C

12.7.1 Creating or Opening a File

12.8 Lets Sum up

12.9 Keywords

12.10 Questions for Review

12.11 Suggested Reading and References

12.12 Answers to Check your Progress

---

## 12.0 OBJECTIVES

---

Understand the Structure in C

Comprehend the Union in C

Understand the concept of Bit Field and Command Line Argument in C

Understand the concept of Files in C

---

## 12.1 INTRODUCTION

---

**Structure** is a user-defined datatype in **C language** which allows us to combine data of different types together. **Structure** helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only.

---

## 12.2 STRUCTURES IN C

---

In C programming language, a structure is a collection of elements of the different data type. The structure is used to create user-defined data type in the C programming language. As the structure used to create a user-defined data type, the structure is also said to be “user-defined data type in C”.

In other words, a structure is a collection of non-homogeneous elements. Using structure we can define new data types called user-defined data types that holds multiple values of the different data type. The formal definition of structure is as follows...

**Structure is a collection of different type of elements under a single name that acts as user defined data type in C.**

Generally, structures are used to define a record in the C programming language. Structures allow us to combine elements of a different data type into a group. The elements that are defined in a structure are called members of structure.

## 12.2.1 HOW TO CREATE STRUCTURE?

To create structure in C, we use the keyword called "struct". We use the following syntax to create structures in c programming language.

```
struct <structure_name>
{
    data_type member1;
    data_type member2, member3;
    .
    .
};
```

Following is the example of creating a structure called Student which is used to hold student record.

### Creating structure in C

```
struct Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
};
```

[Important Points to be remembered

Every structure must terminated with semicolon symbol (;).

"struct" is a keyword, it must be used in lowercase letters only. ]

## 12.2.2 CREATING AND USING STRUCTURE VARIABLES

In a C programming language, there are two ways to create structure variables. We can create structure variable while defining the structure and we can also create after terminating structure using struct keyword. To access members of a structure using structure variable, we use dot (.) operator.

Consider the following example code...

## Creating and Using structure variables in C

```

struct Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
} stud_1 ;           // while defining structure

void main(){
    struct Student stud_2; // using struct keyword

    printf("Enter details of stud_1 : \n");
    printf("Name : ");
    scanf("%s", stud_1.stud_name);
    printf("Roll Number : ");
    scanf("%d", &stud_1.roll_number);
    printf("Percentage : ");
    scanf("%f", &stud_1.percentage);

    printf("***** Student 1 Details *****\n");
    printf("Name of the Student : %s\n", stud_1.stud_name);
    printf("Roll Number of the Student : %i\n", stud_1.roll_number);
    printf("Percentage of the Student : %f\n", stud_1.percentage);
}

```

In the above example program, the structure variable "**stud\_1**" is created while defining the structure and the variable "**stud\_2**" is created using struct keyword. Whenever we access the members of a structure we use the dot (.) operator.

### 12.2.3 MEMORY ALLOCATION OF STRUCTURE

When the structures are used in the C programming language, the memory does not allocate on defining a structure. The memory is allocated when we create the variable of a particular structure. As long as the variable of a structure is created no memory is allocated. The size of memory allocated is equal to the sum of memory required by individual members of that structure. In the above example program, the variables **stud\_1** and **stud\_2** are allocated with 36 bytes of memory each.

## Notes

```
struct Student{
    char stud_name[30];——— 30 bytes
    int roll_number;——— 02 bytes
    float percentage;——— 04 bytes
};                               sum = 36 bytes
```

Here the variable of **Student** structure is allocated with 36 bytes of memory.

[Important Points to be Remembered

All the members of a structure can be used simultaneously.

Until variable of a structure is created no memory is allocated.

The memory required by a structure variable is sum of the memory required by individual members of that structure.]

---

## 12.3 UNIONS IN C

---

In C programming language, the union is a collection of elements of the different data type. The union is used to create user-defined data type in the C programming language. As the union used to create a user-defined data type, the union is also said to be “user-defined data type in C”. In other words, the union is a collection of non-homogeneous elements. Using union we can define new data types called user-defined data types that holds multiple values of the different data type. The formal definition of a union is as follows...

**Union is a collection of different type of elements under a single name that acts as user defined data type in C.**

Generally, unions are used to define a record in the c programming language. Unions allow us to combine elements of a different data type into a group. The elements that are defined in a union are called members of union.

### 12.3.1 HOW TO CREATE UNION?

To create union in C, we use the keyword called "union". We use the following syntax to create unions in C programming language.



```

union <structure_name>
{
    data_type member1;
    data_type member2, member3;
    .
    .
};

```

Following is the example of creating a union called Student which is used to hold student record.

### Creating union in C

```

union Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
};

```

#### Important Points to be Remembered

Every union must terminated with semicolon symbol (;).

"union" is a keyword, it must be used in lowercase letters only.

## 12.3.2 CREATING AND USING UNION

### VARIABLES

In a C programming language, there are two ways to create union variables. We can create union variable while the union is defined and we can also create after terminating union using union keyword. TO access members of a union using union variable, we use dot (.) operator. Consider the following example code...

### Creating and Using union variables in C

```

union Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
} stud_1 ;           // while defining union

void main(){
    union Student stud_2; // using union keyword

    printf("Enter details of stud_1 : \n");
    printf("Name : ");
    scanf("%s", stud_1.stud_name);
    printf("Roll Number : ");
    scanf("%d", &stud_1.roll_number);
    printf("Percentage : ");
    scanf("%f", &stud_1.percentage);

    printf("***** Student 1 Details *****\n");
    printf("Name of the Student : %s\n", stud_1.stud_name);
    printf("Roll Number of the Student : %i\n", stud_1.roll_number);
    printf("Percentage of the Student : %f\n", stud_1.percentage);
}

```

In the above example program, the union variable "**stud\_1**" is created while defining the union and the variable "**stud\_2**" is created using union keyword. Whenever we access the members of a union we use the dot (.) operator.

### 12.3.3 MEMORY ALLOCATION OF UNION

When the unions are used in the C programming language, the memory does not allocate on defining union. The memory is allocated when we create the variable of a particular union. As long as the variable of a union is created no memory is allocated. The size of memory allocated is equal to the maximum memory required by an individual member among all members of that union. In the above example program, the variables **stud\_1** and **stud\_2** are allocated with 30 bytes of memory each.

```

union Student{
    char stud_name[30]; ———— 30 bytes
    int roll_number; ———— 02 bytes
    float percentage; ———— 04 bytes
};                               max = 30 bytes

```

Here the variable of **Student** union is allocated with 30 bytes of memory and it is shared by all the members of that union.

**Check your Progress-1**

1. How to create structure?

---

---

---

2. Define Union

---

---

---

3. Explain memory allocation of union

---

---

---

---

## 12.4 BIT FIELDS IN C

---

When we use structures in the C programming language, the memory required by structure variable is the sum of memory required by all individual members of that structure. To save memory or to restrict memory of members of structure we use **bitfield** concept. Using bitfield we can specify the memory to be allocated for individual members of a structure. To understand the bitfields, let us consider the following example code...

### Date structure in C

```
struct Date
{
    unsigned int day;
    unsigned int month;
    unsigned int year;
};
```

## Notes

Here, the variable of Date structure allocates **6 bytes** of memory.

In the above example structure the members day and month both does not requires 2 bytes of memory for each. Becuase member **day** stores values from **1 to 31** only which requires 5 bits of memory, and the member **month** stores values from **1 to 12** only which required 4 bits of memory. So, to save the memory we use the bitfields.

Consider the following structure with bitfields...

```
Date structure in C

struct Date
{
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year;
};
```

Here, the variable of Date structure allocates **4 bytes** of memory.

---

## 12.5 COMMAND LINE ARGUMENTS IN C

---

In C programming language, command line arguments are the data values that are passed from command line to our program. Using command line arguments we can control the program execution from the outside of the program. Generally, all the command line arguments are handled by the main() method. Generally, the command line arguments can be understood as follows...

**Command line arguments are the parameters passing to main() method from the command line.**

When command line arguments are passed main() method receives them with the help of two formal parameters and they are,

- **int argc**
- **char \*argv[ ]**

**int argc** - It is an integer argument used to store the count of command line arguments are passed from the command line.

**char \*argv[ ]** - It is a character pointer array used to store the actual values of command line arguments are passed from the command line.

### Important Points to be Remembered

- The command line arguments are separated with **SPACE**.
- Always the first command line argument is file path.
- Only string values can be passed as command line arguments
- All the command line arguments are stored in a character pointer array called **argv[ ]**.
- Total count of command line arguments including file path argument is stored in a integer parameter called **argc**.

Consider the following example program...

### Example Program to illustrate command line arguments in C.

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

int main(int argc, char *argv[]){

    int i;
    clrscr() ;

    if(argc == 1){
        printf("Please provide command line argument
s!!!");
        return 0;
    }
    else{
        printf("Total number of arguments are - %d and t
hey are\n\n", argc);
        for(i=0; i<argc ; i++){
            printf("%d -- %s \n", i+1, argv[i]);
        }
        return 0;
    }
}
```

When execute the above program by passing "**Hello welcome to www.btechsmartclass.com**" as command line arguments it produce the following output.

## Notes

```
"C:\Users\User\Desktop\New folder\Command_Line_Arguments\bin\Debug\Command_Line_Arguments.exe" hello welcome to www.btechsmartclass.com
Total number of arguments are - 5 and they are
1 -- C:\Users\User\Desktop\New folder\Command_Line_Arguments\bin\Debug\Command_Line_Arguments.exe
2 -- hello
3 -- welcome
4 -- to
5 -- www.btechsmartclass.com
Process returned 0 (0x0)   execution time : 0.052 s
Press any key to continue.
```

In the above example program we are passing 4 string arguments (Hello, welcome, to and www.btechsmartclass.com) but the default first argument is a file path. So, the total count of command line arguments is 5.

Whenever we want to pass numerical values as command line arguments, they are passed as string values only and we need to convert them into numerical values in the program. Consider the following program that calculates the sum of all command line arguments and displays the result.

### Example Program to display sum of all command line arguments in C.

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

int main(int argc, char *argv[]){

    int i, n, sum = 0;
    clrscr() ;

    if(argc == 1){
        printf("Please provide command line argument s!!!");
        return 0;
    }
    else{
        printf("Total number of arguments are - %d and s
um of those is ", argc);
        for(i=0; i<argc ; i++){
            n = atoi(argv[i]);
            sum += n;
        }
        printf("%d\n", sum);
        return 0;
    }
}
```

When execute the above program by passing "10 20 30 40 50" as command line arguments it produce the following output.

```
"C:\Users\User\Desktop\New folder\Command_line_arguments_2\bin\Debug\Command_line_arguments_2.exe" 10 20 30 40 50
Total number of arguments are - 6 and sum of those is 150
Process returned 0 (0x0)   execution time : 0.101 s
Press any key to continue.
```

In the above example program we are passing 5 string arguments (10, 20, 30 40 and 50). They are converted into integer values by using **atoi()** method which is available in **stdlib.h** header file.

---

## 12.6 FILES IN C

---

Generally, a file is used to store user data in a computer. In other words, computer stores the data using files. we can define a file as follows...

**File is a collection of data that stored on secondary memory like haddisk of a computer.**

C programming language supports two types of files and they are as follows...

- **Text Files (or) ASCII Files**
- **Binary Files**

*Text File (or) ASCII File* - The file that contains ASCII codes of data like digits, alphabets and symbols is called text file (or) ASCII file.

*Binary File* - The file that contains data in the form of bytes (0's and 1's) is called as binary file. Generally, the binary files are compiled version of text files.

### File Operations in C

The following are the operations performed on files in C programming language...

- **Creating (or) Opening a file**
- **Reading data from a file**
- **Writing data into a file**
- **Closing a file**

All the above operations are performed using file handling functions available in C. We discuss file handling functions in the next topic.

---

## 12.7 FILE HANDLING FUNCTIONS IN C

---

File is a collection of data that stored on secondary memory like hard disk of a computer.

The following are the operations performed on files in the C programming language...

- **Creating (or) Opening a file**
- **Reading data from a file**
- **Writing data into a file**
- **Closing a file**

All the above operations are performed using file handling functions available in C.

### 12.7.1 CREATING (OR) OPENING A FILE

To create a new file or open an existing file, we need to create a file pointer of FILE type. Following is the sample code for creating file pointer.

```
File *f_ptr ;
```

We use the pre-defined method **fopen()** to create a new file or to open an existing file. There are different modes in which a file can be opened.

Consider the following code...

```
File *f_ptr ;
```

```
*f_ptr = fopen("abc.txt", "w") ;
```

The above example code creates a new file called **abc.txt** if it does not exists otherwise it is opened in writing mode.

In C programming language, there different modes are available to open a file and they are shown in the following table.

S. No.	Mode	Description
1	r	Opens a text file in <b>reading</b> mode.



S. No.	Mode	Description
2	w	Opens a text file in <b>writing</b> mode.
3	a	Opens a text file in <b>append</b> mode.
4	r+	Opens a text file in both <b>reading and writing</b> mode.
5	w+	Opens a text file in both <b>reading and writing</b> mode. It set the cursor position to the begining of the file if it exists.
6	a+	Opens a text file in both <b>reading and writing</b> mode. The reading operation is performed from begining and writing operation is performed at the end of the file.

**Note** - The above modes are used with text files only. If we want to work with binary files we use **rb, wb, ab, rb+, wb+ and ab+**.

### Check your Progress-2

4. Define – a. int argc

b. char \*argv[ ]

---



---



---

5. Explain the concept of Files

---



---



---

6. State and define two types of files supported byC programming language

---



---



---

---

## 12.8 LET US SUM UP

---

The structure is used to create user-defined data type in the C programming language. The union is used to create user-defined data type in the C programming language. Using bitfield we can specify the memory to be allocated for individual members of a structure. Using command line arguments we can control the program execution from the outside of the program.

---

## 12.9 KEYWORDS

---

**non-homogeneous** elements - The **non-homogeneous** data structures are the one in which the data **elements** doesn't belong to the same data type

**Memory allocation** is a process by which computer programs and services are assigned with physical or virtual **memory** space.

**Command line arguments-** are the data values that are passed from command line to our program

---

## 12.10 QUESTIONS FOR REVIEW

---

1. Discuss creating and using structure variables.
2. Explain how to create union.
3. What is Bit Field in C?
4. Explain Command line argument with example.
5. Enumerate creating (or) opening a file

---

## 12.11 SUGGESTED READINGS AND REFERENCES

---

10. B. Gottfried: Programming with C , Tata McGraw-Hill Edition 2002.

11. E. Balagurusamy : Programming in ANSI C, Tata Mcgraw Hill - Edition 2002.
12. Brain W. Kernighan & Dennis M. Ritchie, The C Programme Language, 2nd Edition (ANSI features) , Prentice Hall 1989.
13. Let Us C- Y.P. Kanetkar, BPB Publication - 2002.
14. Analysis of Numerical Methods—Isacsons& Keller.
15. Numerical solutions of Ord. Diff. Equations—M K Jain
16. Numerical solutions of Partial Diff. Equations—G D Smith.
17. Programming with C, B. Gottfried, Tata-McGraw Hill
18. Programming with C, K. R. Venugopal and Sudeep R. Prasad, Tata-McGraw Hill

---

## **12.12 ANSWERS TO CHECK YOUR PROGRESS**

---

1. Provide explanation with example – 12.2.1
2. Provide definition – 12.3
3. Provide explanation – 12.3.3
4. Provide definition – 12.4
5. Provide definition – 12.6
6. Provide definition – 12.6

---

# UNIT-13 FILE HANDLING FUNCTION AND ERROR HANDLING IN C

---

## STRUCTURE

13.0 Objectives

13.1 Introduction

13.2 Reading from a file

13.2.1 getc()

13.2.2 getw()

13.2.3 fscanf()

13.2.4 fgets()

13.2.5 fread()

13.3 Writing into a File

13.3.1 putc()

13.3.2 putw()

13.3.3 fprintf()

13.3.4 fputs()

13.3.5 fwrite()

13.4 Closing a File

13.5 Cursor Positioning Functions in Files

13.5.1 ftell()

13.5.2 rewind()

13.5.3 fseek()

13.6 Let us sum up

13.7 Keywords

13.8 Questions for Review

13.9 Suggested Reading and References

---

## 13.0 OBJECTIVES

---

Understand the concept of File Handling Functions in C like reading, writing and closing of a file.

Enumerate Cursor positing function in Files

---

## 13.1 INTRODUCTION

---

A file is used to store user data in a computer. File is a collection of data that stored on secondary memory like hard disk of a computer. The following are the operations performed on files in the C programming language...

---

## 13.2 READING FROM A FILE

---

The reading from a file operation is performed using the following pre-defined file handling methods.

getc()

getw()

fscanf()

fgets()

fread()

**13.2.1** *getc( \*file\_pointer )* - This function is used to read a character from specified file which is opened in reading mode. It reads from the current position of the cursor. After reading the character the cursor will be at next character.

## Notes

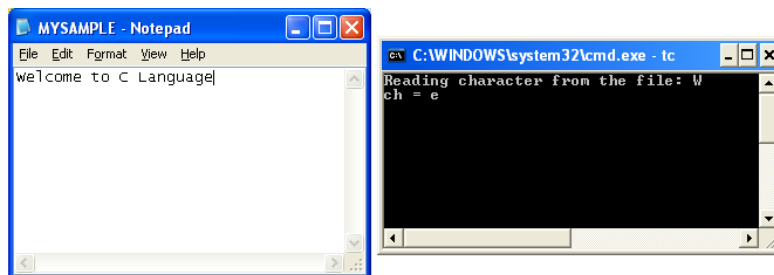
### Example Program to illustrate `getc()` in C.

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

int main(){

    FILE *fp;
    char ch;
    clrscr();
    fp = fopen("MySample.txt","r");
    printf("Reading character from the file: %c\n",
    getc(fp));
    ch = getc(fp);
    printf("ch = %c", ch);
    fclose(fp);
    getch();
    return 0;
}
```

### Output



**13.2.2 `getw( *file_pointer )`** - This function is used to read an integer value from the specified file which is opened in reading mode. If the data in file is set of characters then it reads ASCII values of those characters.

### Example Program to illustrate `getw()` in C.

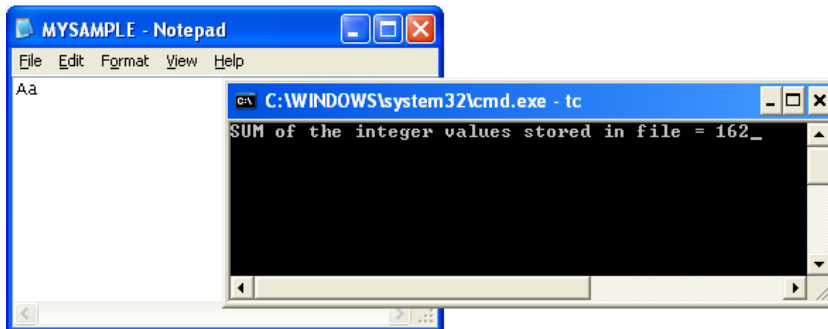
```
#include<stdio.h>
#include<conio.h>

int main(){

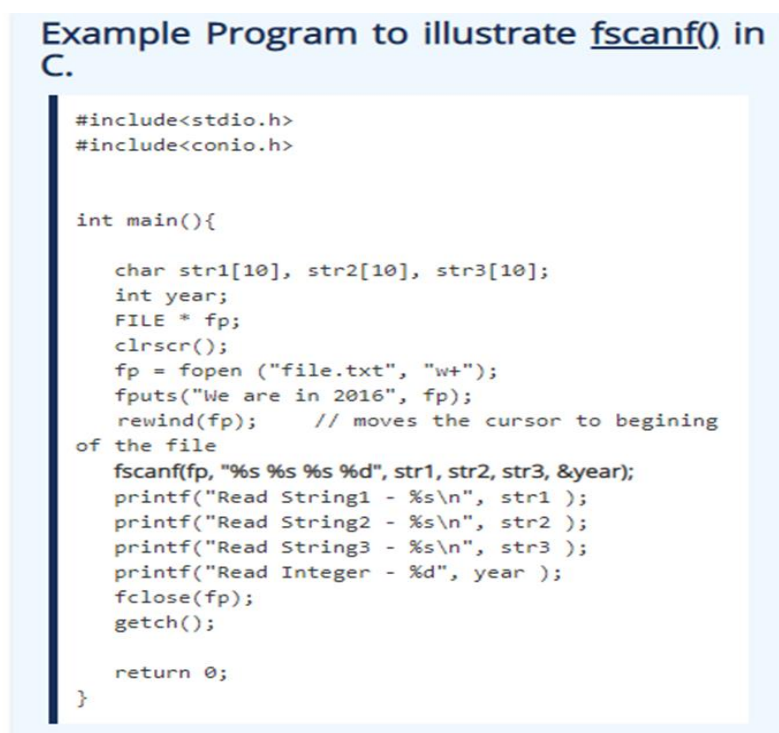
    FILE *fp;
    int i,j;
    clrscr();
    fp = fopen("MySample.txt","w");
    putw(65,fp); // inserts A
    putw(97,fp); // inserts a
    fclose(fp);
    fp = fopen("MySample.txt","r");
    i = getw(fp); // reads 65 - ASCII value of A
    j = getw(fp); // reads 97 - ASCII value of a
    printf("SUM of the integer values stored in file = %d", i+j); // 65 + 97 = 162
    fclose(fp);
    getch();

    return 0;
}
```

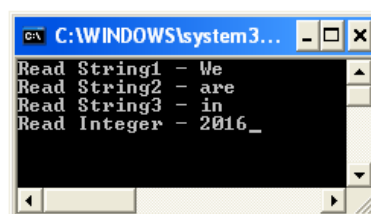
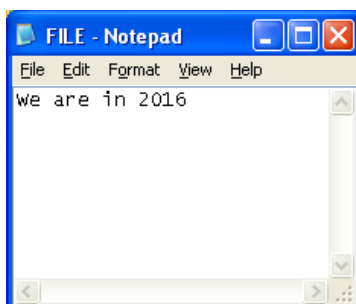
## Output



**13.2.3 fscanf( \*file\_pointer, typeSpecifier, &variableName )** - This function is used to read multiple datatype values from specified file which is opened in reading mode.



## Output:



**13.2.4 fgets( variableName, numberOfCharacters, \*file\_pointer )** - This method is used for reading a set of characters from a file which is

## Notes

opened in reading mode starting from the current cursor position. The `fgets()` function reading terminates with reading NULL character.

### Example Program to illustrate `fgets()` in C.

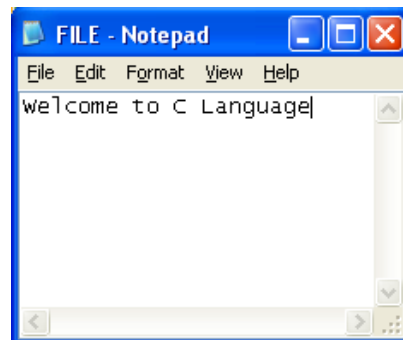
```
#include<stdio.h>
#include<conio.h>

int main(){

    FILE *fp;
    char *str;
    clrscr();
    fp = fopen ("file.txt", "r");
    fgets(str,6,fp);
    printf("str = %s", str);
    fclose(fp);
    getch();

    return 0;
}
```

## Output



**13.2.5 fread( source, sizeofReadingElement, numberOfCharacters, FILE \*pointer )** - This function is used to read specific number of sequence of characters from the specified file which is opened in reading mode.

### Example Program to illustrate `fread()` in C.

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

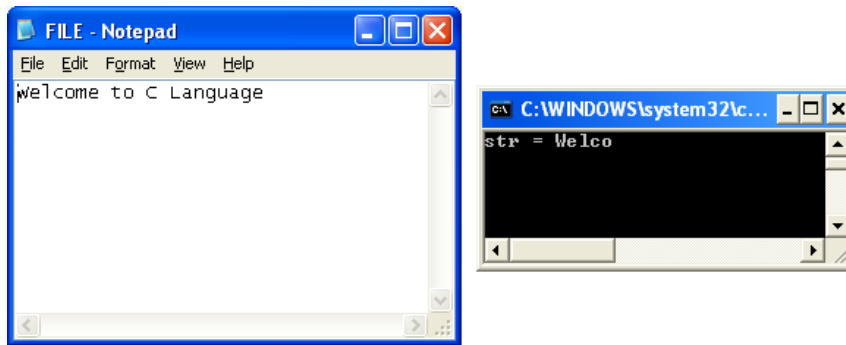
int main(){

    FILE *fp;
    char *str;
    clrscr();
    fp = fopen ("file.txt", "r");
    fread(str,sizeof(char),5,fp);
    str[strlen(str)+1] = 0;
    printf("str = %s", str);
    fclose(fp);
    getch();

    return 0;
}
```



## Output



---

## 13.3 WRITING INTO A FILE

---

The writing into a file operation is performed using the following pre-defined file handling methods.

1. **putc()**
2. **putw()**
3. **fprintf()**
4. **fputs()**
5. **fwrite()**

**13.3.1 putc( char, \*file\_pointer )** - This function is used to write/insert a character to the specified file when the file is opened in writing mode.

### Example Program to illustrate putc() in C.

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

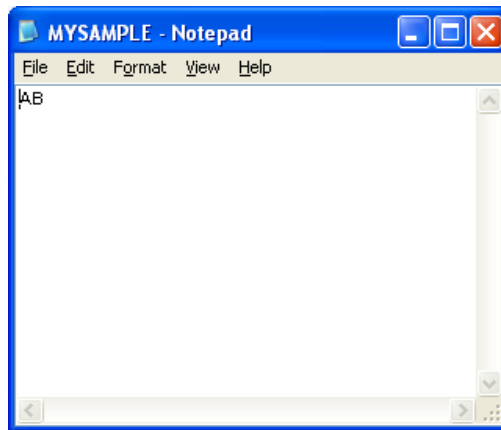
int main(){

    FILE *fp;
    char ch;
    clrscr();
    fp = fopen("C:/TC/EXAMPLES/MySample.txt","w");
    putc('A',fp);
    ch = 'B';
    putc(ch,fp);
    fclose(fp);
    getch();

    return 0;
}
```

## Output

## Notes



**13.3.2 putw( int, \*file\_pointer )** - This function is used to writes/inserts an integer value to the specified file when the file is opened in writing mode.

```
Example Program to illustrate putw() in C.

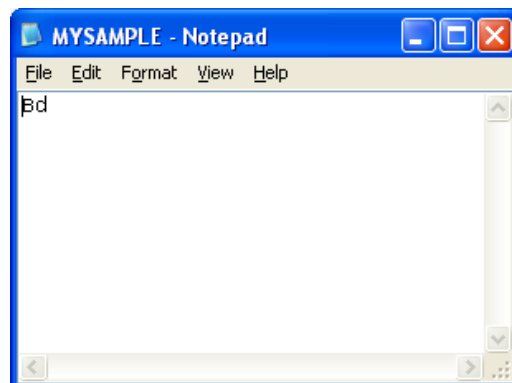
#include<stdio.h>
#include<conio.h>

int main(){

    FILE *fp;
    int i;
    clrscr();
    fp = fopen("MySample.txt","w");
    putw(66,fp);
    i = 100;
    putw(i,fp);
    fclose(fp);
    getch();

    return 0;
}
```

### Output



**13.3.3 fprintf( \*file\_pointer, "text" )** - This function is used to writes/inserts multiple lines of text with mixed data types (char, int, float, double) into specified file which is opened in writing mode.

### Example Program to illustrate `fprintf()` in C.

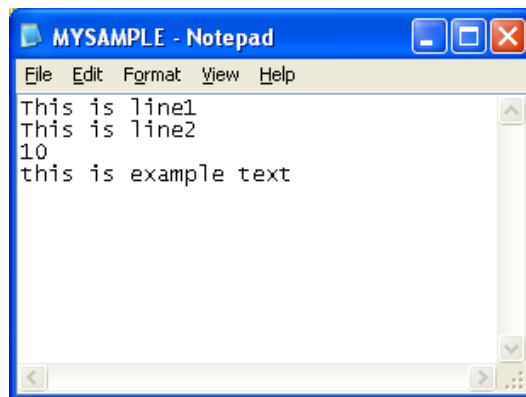
```
#include<stdio.h>
#include<conio.h>

int main(){

    FILE *fp;
    char *text = "\nthis is example text";
    int i = 10;
    clrscr();
    fp = fopen("MySample.txt","w");
    fprintf(fp,"This is line1\nThis is line2\n%d", i);
    fprintf(fp,text);
    fclose(fp);
    getch();

    return 0;
}
```

Output



```
MYSAMPLE - Notepad
File Edit Format View Help
This is line1
This is line2
10
this is example text
```

**13.3.4 `fputs( "string", *file_pointer )`** - This method is used to insert string data into specified file which is opened in writing mode.

### Example Program to illustrate `fputs()` in C.

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

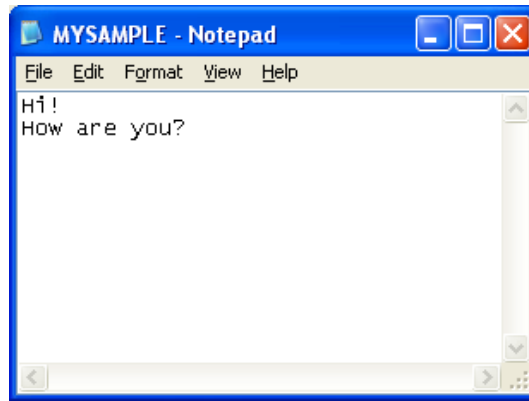
int main(){

    FILE *fp;
    char *text = "\nthis is example text";
    clrscr();
    fp = fopen("MySample.txt","w");
    fputs("Hi!\nHow are you?",fp);
    fclose(fp);
    getch();

    return 0;
}
```

## Notes

Output:



13.3.5 `fwrite( "StringData", sizeof(char), numberOfCharacters, FILE *pointer )` - This function is used to insert specified number of characters into a binary file which is opened in writing mode.

Example Program to illustrate `fwrite()` in C.

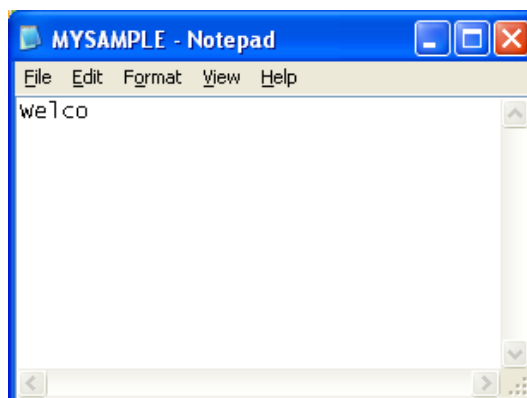
```
#include<stdio.h>
#include<conio.h>

int main(){

    FILE *fp;
    char *text = "Welcome to C Language";
    clrscr();
    fp = fopen("MySample.txt","wb");
    fwrite(text,sizeof(char),5,fp);
    fclose(fp);
    getch();

    return 0;
}
```

Output



---

## 13.4 CLOSING A FILE

---

Closing a file is performed using a pre-defined method `fclose()`.

```
fclose( *f_ptr )
```

The method `fclose()` returns '0' on success of file close otherwise it returns EOF (End Of File).

### Check your Progress-1

1. Explain `getw( *file_pointer )`

---

---

---

2. State `putw( int, *file_pointer )`

---

---

---

3. What do you understand by Closing a file

---

---

---

---

## 13.5 CURSOR POSITIONING FUNCTION IN C

---

C programming language provides various pre-defined methods to set the cursor position in files. The following are the methods available in C, to position cursor in a file.

1. `ftell()`
2. `rewind()`
3. `fseek()`

**13.5.1 `ftell( *file_pointer )`** - This function returns the current position of the cursor in the file.

## Notes

### Example Program to illustrate ftell() in C.

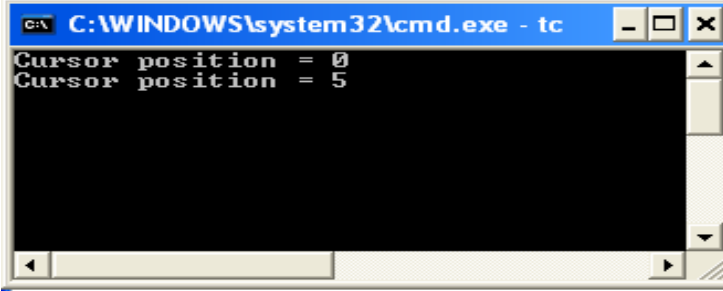
```
#include<stdio.h>
#include<conio.h>

int main(){

    FILE *fp;
    int position;
    clrscr();
    fp = fopen ("file.txt", "r");
    position = ftell(fp);
    printf("Cursor position = %d\n",position);
    fseek(fp,5,0);
    position = ftell(fp);
    printf("Cursor position = %d", position);
    fclose(fp);
    getch();

    return 0;
}
```

### Output



```
C:\WINDOWS\system32\cmd.exe - tc
Cursor position = 0
Cursor position = 5
```

**13.5.2 rewind( \*file\_pointer )** - This function is used reset the cursor position to the beginning of the file.

### Example Program to illustrate rewind() in C.

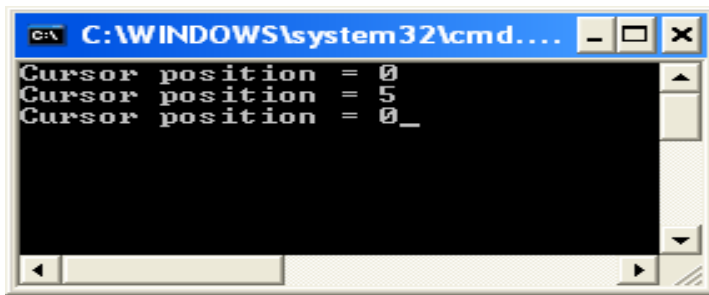
```
#include<stdio.h>
#include<conio.h>

int main(){

    FILE *fp;
    int position;
    clrscr();
    fp = fopen ("file.txt", "r");
    position = ftell(fp);
    printf("Cursor position = %d\n",position);
    fseek(fp,5,0);
    position = ftell(fp);
    printf("Cursor position = %d\n", position);
    rewind(fp);
    position = ftell(fp);
    printf("Cursor position = %d", position);
    fclose(fp);
    getch();

    return 0;
}
```

Output:



```
C:\WINDOWS\system32\cmd...
Cursor position = 0
Cursor position = 5
Cursor position = 0_
```

### 13.5.3 seek( \*file\_pointer, numberOfCharacters, fromPosition ) -

This function is used to set the cursor position to the specific position. Using this function we can set the cursor position from three different position they are as follows.

- from beginning of the file (indicated with 0)
- from current cursor position (indicated with 1)
- from ending of the file (indicated with 2)

#### Example Program to illustrate fseek() in C.

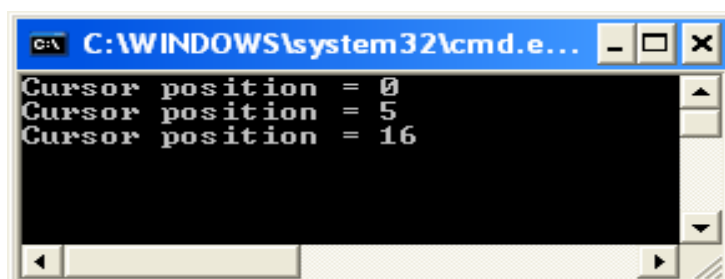
```
#include<stdio.h>
#include<conio.h>

int main(){

    FILE *fp;
    int position;
    clrscr();
    fp = fopen ("file.txt", "r");
    position = ftell(fp);
    printf("Cursor position = %d\n",position);
    fseek(fp,5,0);
    position = ftell(fp);
    printf("Cursor position = %d\n", position);
    fseek(fp, -5, 2);
    position = ftell(fp);
    printf("Cursor position = %d", position);
    fclose(fp);
    getch();

    return 0;
}
```

Output:



```
C:\WINDOWS\system32\cmd.e...
Cursor position = 0
Cursor position = 5
Cursor position = 16
```

---

## 13.6 ERROR HANDLING IN C

---

C programming language does not support error handling that are occurred at program execution time. However, C provides a header file called **error.h**. The header file **error.h** contains few methods and variables that are used to locate error occurred during the program execution. Generally, C programming function returns **NULL** or **-1** in case of any error occurred, and there is a global variable called **errno** which stores the error code or error number. The following table lists few errno values and their meaning.

Error Number	Meaning
1	Specified operation not permitted
2	No such file or directory.
3	No such process.
4	Interrupted system call.
5	IO Error
6	No such device or address
7	Argument list too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

C programming language provides the following two methods to represent errors occurred during program execution.



- **perror()**
- **strerror()**

**perror()** - The perror() function returns a string passed to it along with the textual representation of current errno value.

**strerror()** - The strerror() function returns a pointer to the string representation of the current errno value. This method is defined in the header file **string.h**

Consider the following example program...

### Example Program to illustrate error handling in C.

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

int main(){

    FILE *f_ptr;

    f_ptr = fopen("abc.txt", "r");

    if(f_ptr == NULL){
        printf("Value of errno: %d\n ", errno);
        printf("The error message is : %s\n", strerror(e
rrno));
        perror("Message from perror");
    }
    else{
        printf("File is opened in reading mode!");
        fclose(f_ptr);
    }

    return 0;
}
```

Output:

```
"C:\Users\User\Desktop\New folder\Error_Handling\bin\Debug\Error_Handling.exe"
Value of errno: 2
The error message is : No such file or directory
Message from perror: No such file or directory

Process returned 0 (0x0) execution time : 0.108 s
Press any key to continue.
```

### Check your Progress-1

4. Explain seek( \*file\_pointer, numberOfCharacters, fromPosition )

---



---



---

5. Define perror( ) & strerror( )

---

---

---

---

### 13.7 LET US SUM UP

---

We came across different file handling functions like reading, writing and closing a file. C provides a header file called **error.h**. The header file **error.h** contains few methods and variables that are used to locate error occurred during the program execution.

---

### 13.8 KEYWORDS

---

**Mode-** a way or manner in which something occurs or is experienced, expressed, or done.

**Predefined** - defined, limited, or established in advance.

**Execution time** - The **execution time** or **CPU time** of a given task is **defined** as the **time** spent by the system executing that task, including the **time** spent executing run-**time** or system services on its behalf.

---

### 13.9 QUESTIONS FOR REVIEW

---

1. Discuss 2 reading function in details
2. What is writing into a file? Explain with the help of 3 functions with examples
3. Discuss - C programming language provides various pre-defined methods to set the cursor position in files
4. Explain Error Handling in C

---

### 13.10 SUGGESTED READINGS AND REFERENCES

---

1. B. Gottfried: Programming with C , Tata McGraw-Hill Edition 2002.
2. E. Balagurusamy : Programming in ANSI C, Tata Mcgraw Hill - Edition 2002.
3. Brain W. Kernighan & Dennis M. Ritchie, The C Programme Language, 2nd Edition (ANSI features) , Prentice Hall 1989.
4. Let Us C- Y.P. Kanetkar, BPB Publication - 2002.
5. Analysis of Numerical Methods—Isacsons& Keller.
6. Numerical solutions of Ord. Diff. Equations—M K Jain
7. Numerical solutions of Partial Diff. Equations—G D Smith.
8. Programming with C, B. Gottfried, Tata-McGraw Hill
9. Programming with C, K. R. Venugopal and Sudeep R. Prasad, Tata-McGraw Hill

---

## **13.11 ANSWERS TO CHECK YOUR PROGRESS**

---

1. Provide explanation with example – 13.2.2
2. Provide explanation with example – 13.3.2
3. Provide explanation – 13.4
4. Provide explanation with example – 13.5.3
5. Provide definition – 13.6

---

# UNIT 14: APPLICATION OF C IN NUMERICAL ANALYSIS

---

## STRUCTURE

14.0 Objectives

14.1 Introduction

14.2 Lagrange Interpolation

14.3 INTEGRATION -Simpsons 3/8 Rule

14.4 Matrix inversion: Gauss Jordan method.

14.5 Largest Eigen value and corresponding eigen vector of a square matrix: Power method.

14.6 System of Linear equation: Gauss Seidal method.

14.7 Let us sum up

14.8 Keywords

14.8 Questions for Review

14.10 Suggested Reading and References

14.11 Answers to Check your Progress

---

## 14.0 OBJECTIVES

---

Understand the practical application of C programming in Numerical analysis.

---

## 14.1 INTRODUCTION

---

Numerical analysis is the study of algorithms that use a numerical approximation to solve complex mathematical and scientific problems. The Numerical methods can only deliver approximate solutions to problems over a defined interval such as time or distance.

These methods in Numerical analysis offer approximate solutions to the problems over a defined interval such as time or distance.

---

## 14.2 LAGRANGE INTERPOLATION

---

Lagrange Interpolation is a process of estimation of an unknown data by analyzing the given reference data known as Lagrange Interpolation.

For given data, (say 'y' at various 'x' in tabulated form), the Lagrange 'y' value corresponding to 'x' values can be found by Lagrange Interpolation.

As the Lagrange Interpolation program is executed, it first asks analyse number of known data.

Then, values of x and corresponding y are asked. In Lagrange interpolation in C language, x and y are defined as arrays so that a number of data can be stored under a single variable name.

After getting value of x and y, the program displays input data so that user can correct any incorrectly input data or re-input some missing data. The user is asked to input the value of 'x' at which the value of 'y' is to be interpolated.

At this step, the value of 'y' is computed in loops using Lagrange interpolation formula.

Lagrange Interpolation =  $f(x) = y_0 + \dots + y_n$

Finally the value of 'y' corresponding to 'x' is found.

At last, user is asked to input '1' to run the Lagrange Interpolation program again.

This is again an Nth degree polynomial approximation formula to the function  $f(x)$ , which is known at discrete points  $x_i$ ,  $i = 0, 1, 2 \dots N$ . The formula can be derived from the Vandermonds determinant but a much simpler way of deriving this is from Newton's divided difference formula. If  $f(x)$  Lagrange is approximated with an Nth degree polynomial then the Nth divided difference of  $f(x)$  constant and (N+1)th divided

## Notes

difference is zero. That is Lagrange Interpolation =  $f[x_0, x_1, \dots, x_n, x] = 0$

### Polynomial Interpolation:

The polynomial interpolation problem is the problem of constructing a polynomial that passes through or interpolates  $n+1$  data points

$(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$

### Lagrange Interpolation:

To construct a polynomial of degree  $n$  passing through  $n+1$  data points

$(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$  we start by

constructing a set basis polynomials  $L_{n,k}(x)$  with the property that

$$L_{n,k}(x_j) = \begin{cases} 1 & \text{when } j = k \\ 0 & \text{when } j \neq k \end{cases}$$

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

```
#include<conio.h>
```

```
#define MaxN 90
```

```
void main()
```

```
{
```

```
float arr_x[MaxN+1], arr_y[MaxN+1], numerator, denominator, x, y=0;

int i, j, n;

clrscr();

printf("Enter the value of n: \n");

scanf("%d", &n);

printf("Enter the values of x and y: \n");

for(i=0; i<=n; i++)

scanf("%f%f", &arr_x[i], &arr_y[i]);

printf("Enter the value of x at which value of y is to be calculated: ");

scanf("%f", &x);

for (i=0; i<=n; i++)          /* loop for finding numerator and
denominator */

{

    numerator=1;

    denominator=1;

    for (j=0; j<=n; j++)

if(j!=i)

{
```

## Notes

```
    numerator *= x-arr_x[j];

    denominator *= arr_x[i]-arr_x[j];

}

y+=(numerator/denominator)*arr_y[i];

}

printf("When x=%4.1f y=%7.1f\n",x,y);

getch();

}
```

Output:

Enter the value of n:5

Enter the values of x and y:

5 150

7 392

11 1452

13 2366

17 5202

Enter the value of x at which value of y is to be calculated: 9

When x=9.0 y=809.9

---

## 14.3 INTEGRATION -SIMPSONS 3/8 RULE

---

### What is Simpsons 3/8 Rule?

The Simpson's 3/8<sup>th</sup> rule was developed by a mathematician named Thomas Simpson. Integration is the process of measuring the area under a function plotted on a graph.



The Simpson's 3/8<sup>th</sup> rule is used in complex numerical integrations. This integration method uses parabolas to approximate each part of the curve. It is basically used to measure an area in a curve.

The Simpson's 3/8<sup>th</sup> method is used for uniformly sampled function integration purpose. The Simpson's 3/8<sup>th</sup> integration method is primarily used for numerical approximation of definite integrals.

### Simpson's Rule Formula

$$\int_a^b f(x) dx \approx \frac{3h}{8} \left[ f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b) \right] = \frac{(b-a)}{8} \left[ f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b) \right]$$

### Algorithm For Simpson's 3/8 Rule

1. Input a, b and the number of intervals.
2. Compute the value of h using this formula:  $h = (b - a) / n$
3. Calculate the value of sum using this formula:  $sum = f(a) + f(b)$
4. If n is an odd number, then
  - sum = sum + 2 \* y(a + i \* h)
  - Else
    - sum = sum + 3 \* y(a + i \* h)
5. Calculate the Simpson's integration value using this formula:  $sum * 3 * h / 8$

### Method 1: C Program For Simpson's 3/8<sup>th</sup> Rule using Function

```
#include<stdio.h>

float function(float temp)
{
    return 1 / (1 + temp * temp);
}

int main()
{
    float lower_bound, upper_bound, h, sum = 0, value;
    int count, interval;
    printf("Enter Lower Boundary Value:\t");
    scanf("%f", &lower_bound);
    printf("Enter Upper Boundary Value:\t");
    scanf("%f", &upper_bound);
    printf("\nEnter Interval Limit:\t");
    scanf("%d", &interval);
    h = (upper_bound - lower_bound) / interval;
    sum = function(lower_bound) + function(upper_bound);
    for(count = 1; count < interval; count++)
    {
        if(count % 3 == 0)
        {
            sum = sum + 2 * function(lower_bound + count * h);
        }
        else
        {
            sum = sum + 3 * function(lower_bound + count * h);
        }
    }
    value = (3 * h / 8) * sum;
    printf("\nValue of Simpson's 3/8 Rule Integration:\t%f\n", value);
    return 0;
}
```

## Output

```

./CodingAlpha
Enter Lower Boundary Value: 1
Enter Upper Boundary Value: 10
Enter Interval Limit: 10
Value of Simpson's 3/8 Rule Integration: 0.687927
Press any key to continue.

```

## Method 2: C Program For Simpsons 3/8 Rule without using Function

```

#include
#include

int main()
{
    float x[10], y[10], sum = 0, h;
    int count, limit, m, n;
    printf("\nEnter Limit:\t");
    scanf("%d", &limit);
    for(count = 0; count < limit; count++)
    {
        printf("\nEnter Value For X[%d]:\t", count);
        scanf("%f", &x[count]);
        printf("\nEnter Value For F{x[%d]}:\t", count);
        scanf("%f", &y[count]);
    }
    h = x[1] - x[0];
    limit = limit - 1;
    sum = sum + y[0];
    for(count = 1; count < limit; count++)
    {
        if(m == 0 || n == 0)
        {
            sum = sum + 3 * y[count];
            if(m == 1)
            {
                n = 1;
            }
            m = 1;
        }
        else
        {
            sum = sum + 2 * y[count];
            m = 0;
            n = 0;
        }
    }
    sum = sum + y[count];
    sum = sum * (3 * h / 8);
    printf("\n\nSimple 3/8 Rule Integral Value:\t%f\n", sum);
    return 0;
}

```

## Output

```

./CodingAlpha
Enter Limit: 2
Enter Value For X[0]: 1
Enter Value For F{x[0]}: 0.25
Enter Value For X[1]: 2
Enter Value For F{x[1]}: 0.45
Simple 3/8 Rule Integral Value: 0.262500
Press any key to continue.

```

**Check your Progress-1**

1. What is Lagrange Interpolation?

---

---

---

2. Explain Simpson's 3/8th rule with algorithm

---

---

---

---

## **14.4 MATRIX INVERSION: GAUSS JORDAN METHOD**

---

The Gauss-Jordan method is used to analyze different systems of linear simultaneous equations that arise in engineering and science. This method finds its application in examining a network under sinusoidal steady state, output of a chemical plant, electronic circuits consisting invariant elements, and more.

The **C program for Gauss-Jordan method** is focused on reducing the system of equations to a diagonal matrix form by row operations such that the solution is obtained directly. Further, it reduces the time and effort invested in back-substitution for finding the unknowns, but requires a little more calculation. (see example)

The Gauss-Jordan method is simply a modification of the Gauss elimination method. The eliminations of the unknowns is performed not only in the equations below, but in those above as well. That is to say – unlike the elimination method, where the unknowns are eliminated from pivotal equation only, this method eliminates the unknown from all the equations.

The program of **Gauss-Jordan Method in C** presented here diagonalizes the given matrix by simple row operations. The additional calculations can be a bit tedious, but this method, overall, can be effectively used for small systems of linear simultaneous equations.

## Notes

In the Gauss-Jordan C program, the given matrix is diagonalized using the following step-wise procedure.

1. The element in the first column and the first row is reduced 1, and then the remaining elements in the first column are made 0 (zero).
2. The element in the second column and the second row is made 1, and then the other elements in the second column are reduced to 0 (zero).
3. Similarly, steps 1 and 2 are repeated for the next 3rd, 4th and following columns and rows.
4. The overall diagonalization procedure is done in a sequential manner, performing only row operations.

### Source Code for Gauss-Jordan Method in C:

```
#include<stdio.h>
int main()
{
    int i,j,k,n;
    float A[20][20],c,x[10];
    printf("\nEnter the size of matrix: ");
    scanf("%d",&n);
    printf("\nEnter the elements of augmented matrix row-wise:\n");
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=(n+1); j++)
        {
            printf(" A[%d][%d]:", i,j);
            scanf("%f",&A[i][j]);
        }
    }
    /* Now finding the elements of diagonal matrix */
    for(j=1; j<=n; j++)
    {
        for(i=1; i<=n; i++)
        {
            if(i!=j)
            {
                c=A[i][j]/A[j][j];
                for(k=1; k<=n+1; k++)
                {
                    A[i][k]=A[i][k]-c*A[j][k];
                }
            }
        }
    }
    printf("\nThe solution is:\n");
    for(i=1; i<=n; i++)
    {
        x[i]=A[i][n+1]/A[i][i];
        printf("\n x%d=%f\n",i,x[i]);
    }
    return(0);
}
```

### Input/Output:

```
Enter the size of matrix: 3
Enter the elements of augmented matrix row-wise:
A[1][1]:10
A[1][2]:-7
A[1][3]:5
A[1][4]:9
A[2][1]:3
A[2][2]:6
A[2][3]:0
A[2][4]:-9
A[3][1]:9
A[3][2]:3
A[3][3]:-2
A[3][4]:-1
The solution is:
x1=0.224806
x2=-1.612403
x3=-0.906977
codewithc.com
```

*Note:* Let us consider a system of 10 linear simultaneous equations. Solving this by Gauss-Jordan method requires a total of 500

multiplication, where that required in the Gauss elimination method is only 333.

Therefore, the Gauss-Jordan method is easier and simpler, but requires 50% more labor in terms of operations than the Gauss elimination method. And hence, for larger systems of such linear simultaneous equations, the Gauss elimination method is the more preferred one

---

## 14.5 EIGEN VALUE: POWER METHOD

---

Power method is used to find the dominant eigen value and the corresponding eigen vector. Eigen value problems generally arise in dynamics problems and structural stability analysis. Power method is generally used to calculate these eigen value and corresponding eigen vector of the given matrix.

The **C program for power method** is just a programming illustration of power method as one of the most well suited iterative approach for machine computations. With this, the numerically greatest eigen value and the subsequent eigen vector can be computed to analyze different engineering problems.

Before going into the program for **Power Method in C** language, let's look at a simple mathematical formulation of eigen values and eigen vector. For this, consider a matrix A. We have to find the column vector

X and the constant L (L=lamda) such that:

$$[A]\{X\} = L\{X\}$$

Now, consider these three set of equations:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = Lx_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = Lx_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = Lx_3$$

These equations can be written as:

$$(a_{11}-L)x_1 + a_{12}x_2 + a_{13}x_3 = 0$$

$$a_{21}x_1 + (a_{22}-L)x_2 + a_{23}x_3 = 0$$

$$a_{31}x_1 + a_{32}x_2 + (a_{33}-L)x_3 = 0$$

## Notes

Now the determinant of the 3\*3 matrix formed of the coefficients of x1, x2 and x3 terms gives three roots, namely L1, L2 and L3 (read L as lamda). These values are called characteristic or eigen values. For each of these values, we get a set of column vector with elements x1, x2 and x3. This vector is the required eigen vector.

The Power method C program given below utilizes continuous approximation of L (lamda) to the eigen value and X to the eigen vector. For this method, the matrix should be symmetric or positive definitive i.e.  $a_{ij} = a_{ji}$ .

### Source Code for Power Method in C:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
    int i,j,n;
    float A[40][40],x[40],z[40],e[40],zmax,emax;
    printf("\nEnter the order of matrix:");
    scanf("%d",&n);
    printf("\nEnter matrix elements row-wise\n");
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        {
            printf("A[%d][%d]=", i,j);
            scanf("%f",&A[i][j]);
        }
    }
    printf("\nEnter the column vector\n");
    for(i=1; i<=n; i++)
    {
        printf("X[%d]=", i);
        scanf("%f",&x[i]);
    }
    do
    {
        for(i=1; i<=n; i++)
        {
            z[i]=0;
            for(j=1; j<=n; j++)
            {
                z[i]=z[i]+A[i][j]*x[j];
            }
        }
    }
```

```
        zmax=fabs(z[1]);
        for(i=2; i<=n; i++)
        {
            if((fabs(z[i]))>zmax)
                zmax=fabs(z[i]);
        }
        for(i=1; i<=n; i++)
        {
            z[i]=z[i]/zmax;
        }
        for(i=1; i<=n; i++)
        {
            e[i]=0;
            e[i]=fabs((fabs(z[i]))-(fabs(x[i])));
        }
        emax=e[1];
        for(i=2; i<=n; i++)
        {
            if(e[i]>emax)
                emax=e[i];
        }
        for(i=1; i<=n; i++)
        {
            x[i]=z[i];
        }
    }
    while(emax>0.001);
    printf("\n The required eigen value is %f",zmax);
    printf("\n\nThe required eigen vector is :\n");
    for(i=1; i<=n; i++)
    {
        printf("%f\t",z[i]);
    }
    getch();
}
```

Input/Output:

```

Enter the order of matrix:3
Enter matrix elements row-wise
a[1][1]=2
a[1][2]=-1
a[1][3]=0
a[2][1]=-1
a[2][2]=2
a[2][3]=-1
a[3][1]=0
a[3][2]=-1
a[3][3]=2
Enter the column vector
x[1]=1
x[2]=0
x[3]=0
The required eigen value is 3.414214
The required eigen vector is :
0.708459 -1.000000 0.705754
codewithc.com

```

---

## 14.6 SYSTEM OF LINEAR EQUATION: GAUSS SEIDEL METHOD

---

### What is Gauss Seidel Method?

The Gauss Seidel method is an iterative process to solve a square system of multiple linear equations. It also is popularly known as **Liebmann method**.

In an iterative method in numerical analysis, every solution attempt is started with an approximate solution of an equation and iteration is performed until the desired accuracy is obtained.

In Gauss-Seidel method, the most recent values are used in successive iterations. The Gauss-Seidel Method allows the user to control round-off error.

The Gauss Seidel method is very similar to Jacobi method and is called as the **method of successive displacement**.

The Gauss Seidel **convergence criteria** depend upon the following two properties:

1. The matrix is diagonally dominant.
2. The matrix is symmetrical and positive – definite.

### Gauss Seidel Method Algorithm

- Step 1: Compute value for all the linear equations for  $X_i$
- Step 2: Initial Array must be available
- Step 3: Compute each  $X_i$  and repeat the above steps
- Step 4: Make use of the absolute relative approximate error after every step to check if the error occurs within a pre-specified tolerance.

## Notes

### Advantages

- Faster iteration process.
- Simple and easy to implement.
- Low on memory requirements.

### Disadvantages

- Slower rate of convergence.
- Requires a large number of iterations to reach the convergence point.

**Note:** This Gauss Seidel method C Program is compiled with GNU GCC compiler using CodeLite IDE on Microsoft Windows 10 operating system.

### Method: Implement Gauss Seidel Method in C Programming

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

int main()
{
    int count, t, limit;
    float temp, error, a, sum = 0;
    float matrix[10][10], y[10], allowed_error;
    printf("\nEnter the Total Number of Equations:\t");
    scanf("%d", &limit);
    printf("Enter Allowed Error:\t");
    scanf("%f", &allowed_error);
    printf("\nEnter the Co-Efficients\n");
    for(count = 1; count <= limit; count++)
    {
        for(t = 1; t <= limit + 1; t++)
        {
            printf("Matrix[%d][%d] = ", count, t);
            scanf("%f", &matrix[count][t]);
        }
    }
    for(count = 1; count <= limit; count++)
    {
        y[count] = 0;
    }
    do
    {
        a = 0;
        for(count = 1; count <= limit; count++)
        {
            sum = 0;
            for(t = 1; t <= limit; t++)
            {
                if(t != count)
                {
                    sum = sum + matrix[count][t] * y[t];
                }
            }

            temp = (matrix[count][limit + 1] - sum) / matrix[count][count];
            error = fabs(y[count] - temp);
            if(error > a)
            {
                a = error;
            }
            y[count] = temp;
            printf("\nY[%d]=\t%f", count, y[count]);
        }
        printf("\n");
    }
    while(a >= allowed_error);
    printf("\n\nSolution\n\n");
    for(count = 1; count <= limit; count++)
    {
        printf("\nY[%d]:\t%f", count, y[count]);
    }
    return 0;
}
```



**Output**

```

./CodingAlpha
Enter the Total Number of Equations: 1
Enter Allowed Error: 0.5

Enter the Co-Efficients
Matrix[1][1] = 1
Matrix[1][2] = 4

Y[1]= 4.000000
Y[1]= 4.000000

Solution

Y[1]: 4.000000
Press any key to continue.

```

**Check your Progress-2**

3. state the steps used for diagonalization in the Gauss-Jordan C program

---



---



---

4. Explain Power Method

---



---



---

5. State advantage and disadvantage of Gauss Seidel Method?

---



---



---



---

## 14.7 LET US SUM UP

---

We came across different concepts of interpolation, integration, matrix and power methods. We understood the application of C code in Numerical analysis with examples

---

## 14.8 KEYWORDS

---

## Notes

**Function** - a **function** is a relation between sets that associates to every element of a first set exactly one element of the second set.

**Approximate** - to estimate a number, amount or total, often. rounding it off to the nearest 10 or 100

**Diagonalization** is the process of transforming a matrix into diagonal form.

**Simultaneous equations** - a set of **equations** in two or more variables for which there are values that can satisfy all the **equations** **simultaneously**

**Iterative methods** - Stationary **iterative** methods solve a **linear** system with an operator approximating the original one; and based on a measurement of the error in the result (the residual), form a "correction equation" for which this process is repeated.

---

## 14.9 QUESTIONS FOR REVIEW

---

1. Write C program for Lagrange Interpolation.
2. State C Program For Simpsons 3/8 Rule without using Function.
3. Discuss Source Code for Gauss-Jordan Method in C
4. Explain Source Code for Power Method in C.
5. How Gauss Seidel Method will be implemented in C Programming?

---

## 14.10 SUGGESTED READINGS AND REFERENCES

---

1. B. Gottfried: Programming with C , Tata McGraw-Hill Edition 2002.
2. E. Balagurusamy : Programming in ANSI C, Tata Mcgraw Hill - Edition 2002.

3. Brain W. Kernighan & Dennis M. Ritchie, The C Programme Language, 2nd Edition (ANSI features) , Prentice Hall 1989.
4. Let Us C- Y.P. Kanetkar, BPB Publication - 2002.
5. Analysis of Numerical Methods—Isacsons& Keller.
6. Numerical solutions of Ord. Diff. Equations—M K Jain
7. Numerical solutions of Partial Diff. Equations—G D Smith.
8. Programming with C, B. Gottfried, Tata-McGraw Hill
9. Programming with C, K. R. Venugopal and Sudeep R. Prasad, Tata-McGraw Hill

---

## **14.11 ANSWERS TO CHECK YOUR PROGRESS**

---

1. Provide explanation – 14.2
2. Provide explanation with algorithm – 14.3
3. Provide steps – 14.4
4. Provide explanation – 14.5
5. Provide advantage and disadvantage – 14.6