

INTRODUCTION TO C PROGRAMMING

Table of Contents

- ✿ Course Objectives
- ✿ Syllabus
- ✿ Course Outcomes (Cos)
- ✿ CO-PO Mapping
- ✿ ??== for each unit
- ✿ Lecture Plan
- ✿ Activity based learning
- ✿ Lecture notes
- ✿ Assignments
- ✿ Part A Q&A
- ✿ Part B Qs
- ✿ List of Supportive online Certification courses
- ✿ Real time applications in day to day life and to industry
- ✿ Contents beyond Syllabus
- ✿ ??=====
- ✿ Assessment Schedule (proposed and actual date)
- ✿ Prescribed Text Books & Reference Books
- ✿ Mini Project Suggestions

Course Objectives

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C

3 0 0 3

OBJECTIVES

- ✿ To develop C Programs using basic programming constructs
- ✿ To develop C programs using arrays and strings
- ✿ To develop applications in C using functions and structures

Syllabus

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C 3 0 0 3

UNIT I INTRODUCTION

9

Structure of C program – Basics: Data Types – Constants – Variables – Keywords – Operators: Precedence and Associativity – Expressions – Input/output statements, Assignment statements – Decision-making statements – Switch statement – Looping statements – Pre-processor directives – Compilation process – Exercise Programs: Check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers

UNIT II ARRAYS

9

Introduction to Arrays – One dimensional arrays: Declaration – Initialization – Accessing elements – Operations: Traversal, Insertion, Deletion, Searching – Two dimensional arrays: Declaration – Initialization – Accessing elements – Operations: Read – Print – Sum – Transpose – Exercise Programs: Print the number of positive and negative values present in the array – Sort the numbers using bubble sort – Find whether the given matrix is diagonal or not.

UNIT III STRINGS

9

Introduction to Strings – Reading and writing a string – String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse – Substring – Insertion – Indexing – Deletion – Replacement – Array of strings – Introduction to Pointers – Pointer operators – Pointer arithmetic – Exercise programs: To find the frequency of a character in a string – To find the number of vowels, consonants and white spaces in a given text – Sorting the names.

UNIT IV FUNCTIONS

9

Introduction to Functions – Types: User-defined and built-in functions – Function prototype – Function definition – Function call – Parameter passing: Pass by value – Pass by reference – Built-in functions (string functions) – Recursive functions – Exercise programs: Calculate the total amount of power consumed by „n” devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)

UNIT V STRUCTURES

9

Introduction to structures – Declaration – Initialization – Accessing the members – Nested Structures – Array of Structures – Structures and functions – Passing an entire structure – Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

TOTAL:45 PERIODS

Course Outcomes

- ✿ CO 1 - Develop algorithmic solutions to simple computational problems K1
- ✿ CO 2 - Read, write, execute by hand simple C programs. K2
- ✿ CO 3 - Structure simple C programs for solving problems using statements K2
- ✿ CO 4 – Represent data using arrays and strings operations K3
- ✿ CO 5 - Decompose a C program into functions and pointers K3
- ✿ CO 6 - Represent and write program using structure and union K3

CO – PO Mapping

CO	PO	Mapping Level	Justification
CO	PO	Mapping Level	Justification
CO1	PO1	2	Apply simple mathematical concepts for writing algorithms
CO1	PO2	2	Identify formulae for the given problem
CO1	PO3	2	Design algorithmic way of problem solving
CO1	PO5	2	Recognize the need of algorithm in implementation
CO1	PO12	1	Apply logic to solve simple problem statement
CO2	PO1	3	Identify the data type and operators to solve the problem
CO2	PO2	3	Design the expression in an efficient way
CO2	PO3	2	Recognize the need of basic data types and operators
CO2	PO5	2	Apply control flow statement for solving the problem
CO2	PO12	1	Formulate the algorithm into executable c code
CO3	PO1	3	Develop a complete program in a simple way
CO3	PO2	3	Recognize the need of control flow statements
CO3	PO3	3	Apply the knowledge to find the possible code for function
CO3	PO5	2	Identify the code for decomposition as function
CO3	PO12	1	Develop functions and reuse it whenever required to reduce the lines of code
CO4	PO1	2	Recognize the need of function concepts
CO4	PO2	2	Apply compound data knowledge to select any one
CO4	PO3	2	Apply the concept of pointers
CO4	PO5	2	Design and Develop program using the selected compound data
CO4	PO12	1	Recognize the need of structure
CO5	PO1	2	Apply the basic idea of handling with union
CO5	PO2	2	Identify the number of modes and operations on structure in detail
CO5	PO3	2	Develop programs using structure and

Lecture Plan

Unit I

Unit I - Introduction

S.No	Topics	No. of Periods	Proposed Date	Actual Lecture Date	Pertaining CO	Taxonomy Level	Mode of Delivery
1	Structure of C program-Basics: Datatypes	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
2	Constants, Variables, Keywords, Operators, Precedence and Associativity	1			CO1	K2	PPT, Chalk & Talk
3	Expressions, Input/Output statements, Assignment statements	1			CO1	K2	PPT, Chalk & Talk
4	Decision-making statements, Switch statement,	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
5.6	Looping statements, Pre-processor directives	2			CO1	K2	PPT, Chalk & Talk
7	Compilation process Exercise Programs- Ex. Prog 1 : Check whether the required amount can be withdrawn based on the available balance	1			CO1	K2	PPT, Chalk & Talk
8,9	Exercise Programs: Ex. Prog. 2 : Menu driven program to find the area of different shapes. Ex. Prog. 3 : Find the sum of even numbers.	2			CO1	K2	PPT, Chalk & Talk

Activity Based Learning

Unit I

Activity Based Learning

- ✿ Learn by solving problems – Tutorial Sessions can be conducted
 - Tutorial sessions available in Skillrack for practice
- ✿ Learn by questioning
- ✿ Learn by doing hands-on IN VIRTUAL LAB.

Lecture Notes

UNIT I INTRODUCTION

9

Structure of C program – Basics: Data Types – Constants –Variables - Keywords – Operators: Precedence and Associativity - Expressions - Input/output statements, Assignment statements –Decision-making statements - Switch statement - Looping statements – Pre-processor directives -Compilation process – Exercise Programs: Check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers

UNIT I

INTRODUCTION

Structure of C Program

A C program is divided into different sections. There are six main sections to a basic c program.

The six sections are,

- Documentation
- Link
- Definition
- Global Declarations
- Main functions
- Sub programs

The whole code follows this outline. Each code has a similar outline. Now let us learn about each of this layer in detail.



Fig1: Basic Structure of a C Program

Documentation Section

The documentation section is the part of the program where the programmer gives the details associated with the program. He usually gives the name of the program, the details of the author and other details like the time of coding and description. It gives anyone reading the code the overview of the code.

Link Section

This part of the code is used to declare all the header files that will be used in the program. This leads to the compiler being told to link the header files to the system libraries.

Definition Section

In this section, we define different constants. The keyword `define` is used in this part.

Example:

```
#define PI= 3.14
```

Global Declaration Section

This part of the code, where the global variables are declared. All the global variable used are declared in this part. The user-defined functions are also declared in this part of the code.

Example:

```
float a (float rd);  
int x;
```

Main Function Section

Every C-programs has the main function. Each main function contains 2 parts. A declaration part and an Execution part. The declaration part is the part where all the variables are declared. The execution part begins with the curly brackets and ends with the curly close bracket. Both the declaration and execution part are inside the curly braces.

Example:

```
void main ()  
{  
float x=10.9;  
printf("%f",x);  
}
```

Sub Program Section

All the user-defined functions are defined in this section of the program.

Example:

```
int sum (int x, int y)
{
Return x+y;
}
```

Sample Program

The C program here will find the area of a square

Example:

File Name: areaofasquare.c

Aim: A C program to find the area of a square (user enters the value of a side)

```
#include<stdio.h>
#include<conio.h>
void main()
{
int side,area;
printf("Enter the value of side");
scanf("%d",&side);
area=side*side;
printf("The area of a Square is %d",area);
getch();
}
```

Basic Data Types

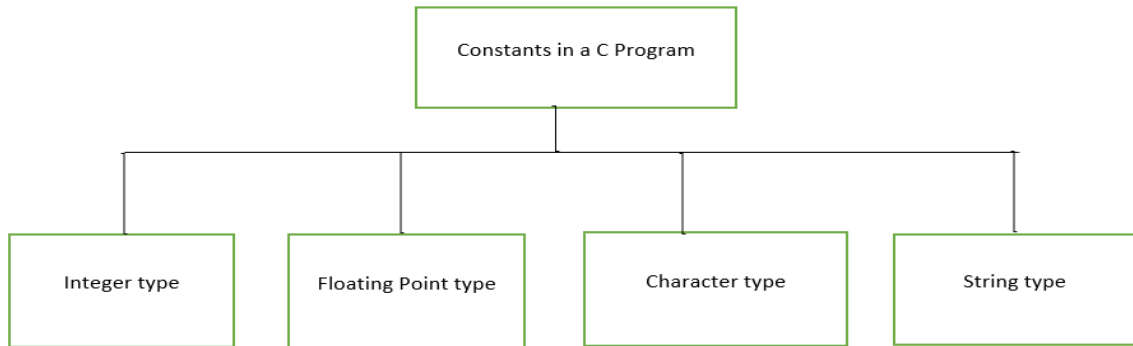
C language provides very few basic datatypes. The datatypes are specified by a standard keyword. The data types are used to define the type of data for particular variable. Various data types that are used in C is enlisted in the following table.

Type	Size	Range
char	1 byte	-127 to 127 or 0 to 255
unsigned	1 byte	0 to 255
signed char	1 byte	-127 to 127
int	4 bytes	-2147483648 to 2147483647
unsigned int	4 bytes	0 to 4294967295
signed int	4 bytes	-2147483648 to 2147483647
short int	2 bytes	-32768 to 32767
unsigned short int	2 bytes	0 to 65535
signed short int	2 bytes	-32768 to 32767
long int	4 bytes	-2147483647 to 2147483647
signed long int	4 bytes	-2147483647 to 2147483647
unsigned long int	4 bytes	0 to 4294967295
float	4 bytes	+/-3.4e +/-38
double	8 bytes	+/-1.7e +/-308
long double	8 bytes	+/-1.7e +/-308

<https://www.youtube.com/watch?v=bS6uNMmloQ0>

Constants:

Constants are identifiers whose value does not change.



Integer type Constant

A constant of integer type consists of a sequence of digits.

Example:

1,34,546,8909 etc. are valid integer constants.

Floating point type constant

Integer numbers are inadequate to express numbers that have a fractional point. A floating point constant therefore consists of an integer part, a decimal point, a fractional part, and an exponent field containing an e or E (e means exponents) followed by an integer where the fraction part and integer part are a sequence of digits.

Example:

Floating point numbers are 0.02, -0.23, 123.345, +0.34 etc.

Character Constant

A character constant consists of a single character enclosed in single quotes. For example, „a“, „@“ are character constants. In computers, characters are stored using machine character set using ASCII codes.

String Constant

A string constant is a sequence of characters enclosed in double quotes. So “a” is not the same as „a“. The characters comprising the string constant are stored in successive memory locations. When a string constant is encountered in a C program, the compiler records the address of the first character and appends a null character („\0“) to the string to mark the end of the string.

Declaring Constant

```
#define PI 3.14159  
#define service_tax 0.12
```

Rules for declaring constant

Rule 1: Constant names are usually written in capital letters to visually distinguish them from other variable names which are normally written in lower case characters

Rule 2: No blank spaces are permitted in between the # symbol and define keyword

Rule 3: Blank space must be used between #define and constant name and constant value

Rule 4: #define is a pre-processor compiler directive and not a statement. Therefore, it does not end with a semi-colon.

Variables:

A variable is defined as a meaningful name given to the data storage location

in computer memory. C language supports two basic kinds of variables

- **Numeric Variable**
- **Character Variable**

Numeric Variable:

Numeric variable can be used to store either integer value or floating point values. While an integer value is a whole number without a fraction part or decimal point a floating point value can have a decimal point.

Numeric variables may also be associated with modifiers like short, long, signed, and unsigned. The difference between signed and unsigned numeric variable is that signed variable can be either negative or positive but unsigned variables can only be positive.

Character Variable:

Character variable can include any letter from the alphabet or from the ASCII chart and numbers 0 – 9 that are given with in single quotes.

Example:

```
int emp_num;  
float salary;  
double balance;
```

In C variable are declared at three basic places as follows

- When a variable is declared inside a function it is known as a local variable.
- When a variable is declared in the definition of function parameter it is known as formal parameter.
- When the variable is declared outside all functions, it is known as a global variable.

Keywords:

Keywords are special reserved words associated with some meaning.

auto	double	int	struct
continue	if	volatile	break
else	long	switch	default
signed	while	case	enum
register	typedef	do	sizeof
char	extern	return	union
for	static	const	float
short	unsigned	goto	void

Operators

C provides a rich set of operators to manipulate data. We can divide all the C operators into the following groups

- Arithmetic Operators
- Unary Operator
- Relational Operators
- Logical Operators
- Assignment Operator
- Bitwise Operators

Arithmetic Operators

The following table list arithmetic operators

Operator	Description	Example
+	Addition	A + B
-	Subtraction	A - B
*	Multiplication	A * B
/	Division	A/B
%	Modulus	A%B

```

/* Example to understand Arithmetic operator */
#include<stdio.h>
#include<conio.h>
void main()
{
int a = 10, b=3;
printf("a + b = ", (a + b) );
printf("a - b = ",(a - b) );
printf("a * b = ",(a * b) );
printf("a / b = ",(a / b) );
printf("a % b = ",(a % b) );
}

```

Output:

```

a + b = 13
a - b = 7
a * b = 30
a / b = 3 a
a% b = 1

```

Unary Operators

The following are the unary operators

Operator	Description	Example
+	Unary plus operator	+A
-	Unary minus operator	-A
++	Increment operator	++A or A++
--	Decrement operator	--A or A--

++ and - - works in two different modes

- Pre increment/decrement – When it is part of a statement, increment/decrement gets evaluated first, followed by the execution of the statement.
- Post increment/decrement – When the operator is part of a statement, the statement gets processed first, followed by increment/decrement operation.

// Example for pre increment/decrement

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a = 10, b=3;
    printf("a++ = ", (a ++) );
    printf("a - - = " , (a - -) );
}
```

Output:

**a++ = 11
b-- = 2**

Relational Operators

Relational operators are used to test condition and results true or false value, The following table lists relational operators

Operator	Description	Example
==	Two values are checked, and if equal, then the condition becomes true	(A == B)
!=	Two values are checked to determine whether they are equal or not, and if not equal, then the condition becomes true	(A != B)
>	Two values are checked and if the value on the left is greater than the value on the right, then the condition becomes true.	(A > B)
<	Two values are checked and if the value on the left is less than the value on the right, then the condition becomes true	(A < B)
>=	Two values are checked and if the value on the left is greater than equal to the value on the right, then the condition becomes true	(A >= B)
<=	Two values are checked and if the value on the left is less than equal to the value on the right, then the condition becomes true	(A <= B)

/* Example to understand Relational operator */

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a = 10, b=20;
printf("a= = b=", (a ==b) );
printf("a !=b= " , (a!=b) );
printf("a>b=", (a>b));
printf("a>=b=", (a>=b));
printf("a<b=", (a<b));
printf("a<=b=", (a<=b))
}
```

Output:

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

Logical Operators

Logical operators are used to combine more than one condition.

The following table lists logical operators

Operator	Description	Example
&&	This is known as Logical AND & it combines two variables or expressions and if and only if both the operands are true, then it will return true	(A && B) is false
	This is known as Logical OR & it combines two variables or expressions and if either one is true or both the operands are true, then it will return true	(A B) is true
!	Called Logical NOT Operator. It reverses the value of a Boolean expression	!(A && B) is true

Example

```
#include<stdio.h>
void main()
{
    boolean a = true;
    boolean b = false;
    printf("a && b = " + (a&&b) );
    printf("a || b = " + (a||b) );
    printf("!(a && b) = " + !(a && b) );
}
```

Output:

a && b = false

a || b = true

!(a && b) = true

Assignment Operator

Simple Assignment

=, assigns right hand side value to left hand side variable

Ex:

```
int a;
```

```
a = 10;
```

Compound Assignment

`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `>>=`, `<<=`, assigns right hand side value after the computation to left hand side variable

Ex:

```
int a;  
int b;  
a += 10; // means a = a + 10;  
a &= b; // means a = a & b;
```

Bitwise Operators

Bitwise operator act on integral operands and perform binary operations. The lists of bitwise operators are

i.	Bitwise AND	&
ii.	Bitwise OR	
iii.	Bitwise EXOR	^
iv.	Bitwise NOT	~ (unary operator)
v.	Shift Left	<<
vi.	Shift Ri	>>

Bitwise AND

The `&` operator compares corresponding bits between two numbers and if both the bits are 1, only then the resultant bit is 1. If either one of the bits is 0, then the resultant bit is 0.

Example :

```
int x = 5; int y = 9; x & y = 1
```

5 - >	0 1 0 1
9 - >	1 0 0 1
	0 0 0 1

Bitwise OR

The `|` operator will set the resulting bit to 1 if either one of them is 1.

It will return 0 only if both the bits are 0.

Example :

```
int x = 5;
```

```
int y = 9;
```

```
x | y = 13
```

5 - >	0 1 0 1
9 - >	1 0 0 1
	1 1 0 1

Bitwise EXOR

The `^` operator compares two bits to check whether these bits are different. If they are different, the result is 1. Otherwise, the result is 0.

This operator is also known as XOR operator.

Example :

```
int x = 5;
```

```
int y = 9;
```

```
x ^ y = 12
```

5 - >	0 1 0 1
9 - >	1 0 0 1
	1 1 1 0

```
#include<stdio.h>
void main()
{
int x = 5;
int y = 9;
int a = x & y; int b = x | y; int c = x ^ y;
printf("x & y = "+a);
printf(" x | y = "+b);
printf("x ^ y = "+c);
}
```

Output:

x & y = 1

x | y = 13

x ^ y = 12

Bitwise NOT

The negation `~` operators complements all the bits, 1 are converted to 0 and 0s are converted to 1s.

For Eg.

```
int a =5;
```

```
~a = -5
```

```
5 -> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
~5 -> 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0
```

Shift Operators

The shift operators (`<<` and `>>`) shift the bits of a number to the left or right, resulting in a new number. They are used only on integral numbers (and not on floating point numbers, i.e. decimals).

Shift Right

The right shift operator(`>>`) is used to divide a number in the multiples of 2, while the left shift operator(`<<`) is used to multiply a number in the multiples of 2.

For Eg.

```
int x = 16; x = x >> 3;
```

right shift operator `>>`, divides by 2 to the power of number specified after the operator. In this case, we have 3 as the value after the right shift operator. So, 16 will be divided by the value 2 to the power of 3, which is 8.

The result is 2.

When we represent 16 in binary form, we will get the following binary value :

0 1 0 0 0 0

When we apply >> which is the right shift operator, bit represented by 1 moves by the positions to the right (represented by the 3 right shift operator). After number after the binary digit 1, we will get : shifting

0 1 0

x = 2



Shift Left

Eg.

```
int x = 8;
```

```
x = x << 4;
```

left shift operator <<, multiplies by 2 to the power of number specified after the operator. In this case, we have 4 as the value after the left shift operator. So, 8 will be multiplied by the value 2 to the power of 4, which is 16.

The result is 128.

When we represent 8 in binary form, we will get the following binary value:

0 1 0 0 0

When we apply << which is the left shift operator, bit represented by 1 moves by the positions to the left (represented by the number right shift 4 After
after the binary digit 1, we will get : operator). shifting

0 1 0 0 0 0 0 0 0

X=128



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

```
Void main()
```

```
{
```

```
int x =8;
```

```
printf("The original value of x is ",x);
```

```
printf("After using << 2, the new value is ",x << 2);
```

```
printf("After using >> 4, the new value is ", x >> 4);
```

```
}
```

Output:

The original value of x is 8

After using << 2, the new value is 2

After using >> 4, the new value is 128

Precedence and Associativity

OP	ASSO	OP	ASSO	OP	ASSO	OP	ASSO
()							
[]		<<	left-to-right	^	left-to-right		
.	left-to-right	>>				? :	right-to-left
->							
++ --							
++ --						=	
+ -		< <				+= -=	
! ~		=				*= /=	
(type		> >	left-to-right		left-to-right	%= &	
)	right-to-left	=				=	
*						^= =	
&						<<=	
sizeof						>>=	right-to-left
		==					
		!=	left-to-right	&&	left-to-right		
* /							
%	left-to-right					,	left-to-right
+ -	left-to-right	&	left-to-right		left-to-right		

Expression

An **expression** is a formula in which operands are linked to each other by the use of operators to compute a value. An operand can be a function reference, a variable, an array element or a constant.

Example: $x = 9/2 + a-b;$

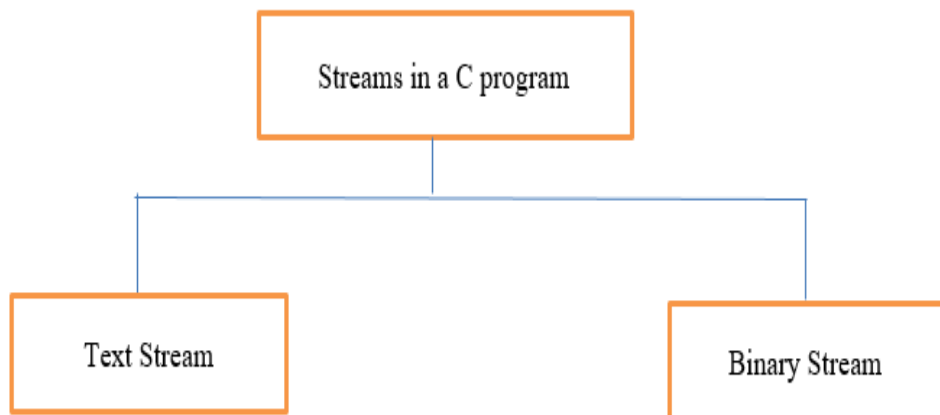
Input /Output Statements:

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

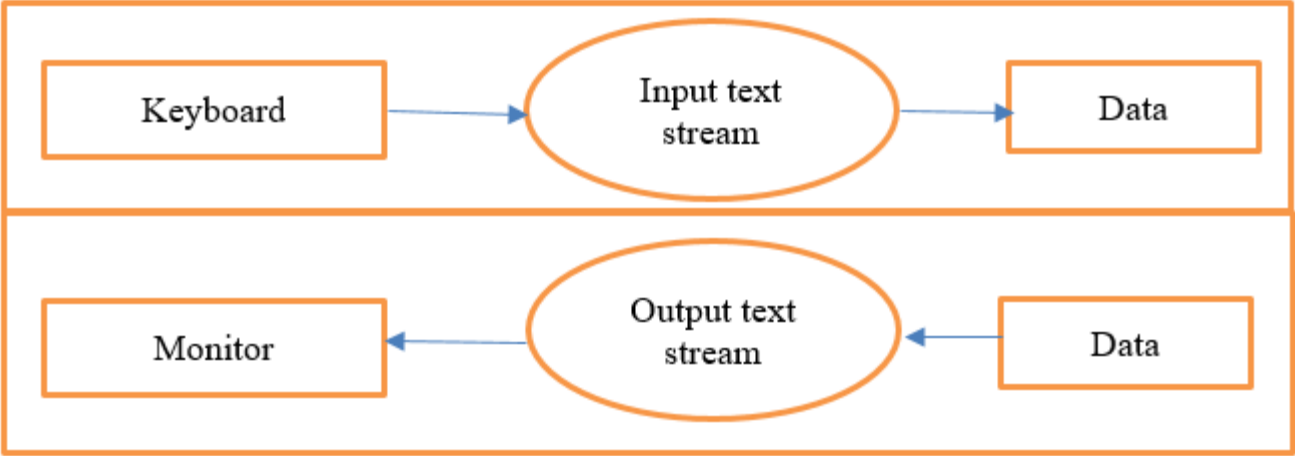
C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

Streams

A stream act in two ways. It is the source of data as well as the destination of the data. C programs input and output data from a stream. It is associated with a physical devices such as the monitor or with a file stored on the secondary memory. C use two forms of streams **Text and Binary**.



We can do input/output from the keyboard from any file. Consider input of data is the keyboard and output data is the monitor.



Printf() and Scanf () functions

The standard input-output header file, named stdio.h contains the definition of the functions printf() and scanf(), which are used to display output on screen and to take input from user respectively.

```
#include<stdio.h>
#include<conio.h>
void main()
{
float i;
printf("Enter the value");
scanf("%f",&i);
printf("The value is %f=",i);
getch();
}
```

Format String	Meaning
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating point number
%c	To scan or print a character
%s	To scan or print a character string.

Putchar() & getchar() functions

The `getchar()` function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. The `putchar()` function displays the character passed to it on the screen and returns the same character.

```
#include<stdio.h>
void main()
{
char q;
Printf("Enter a Character");
q=getchar();
putchar(q);
}
```

Assignment Statement

An **assignment statement** sets the value stored in the storage location denoted by a `variable_name`. In other words, it copies a value into the variable.

Syntax:

```
variable = expression;
```

Decision Making Statement

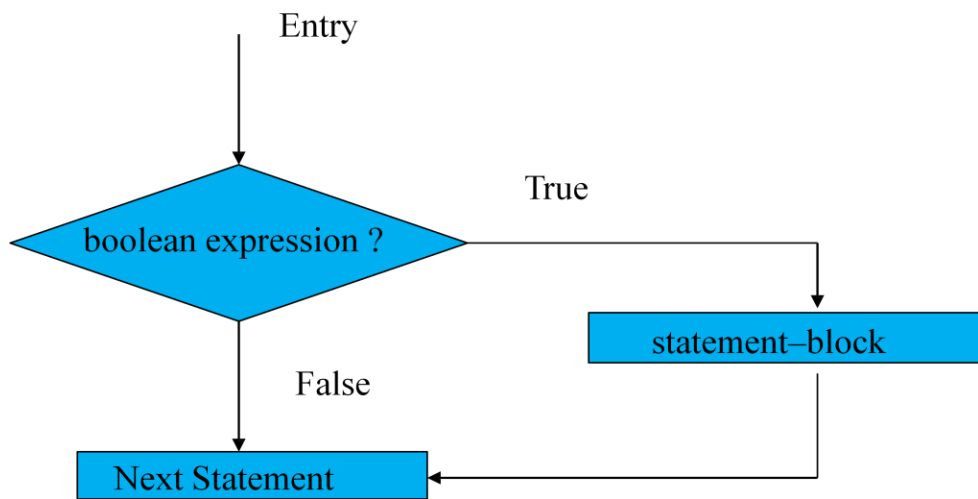
Decision making statements are mainly three type.

- if
- if...else
- if...else...if

Simple if

syntax :

```
if(Booleanexpressio)
{
statement-block;
}
Next statement;
```



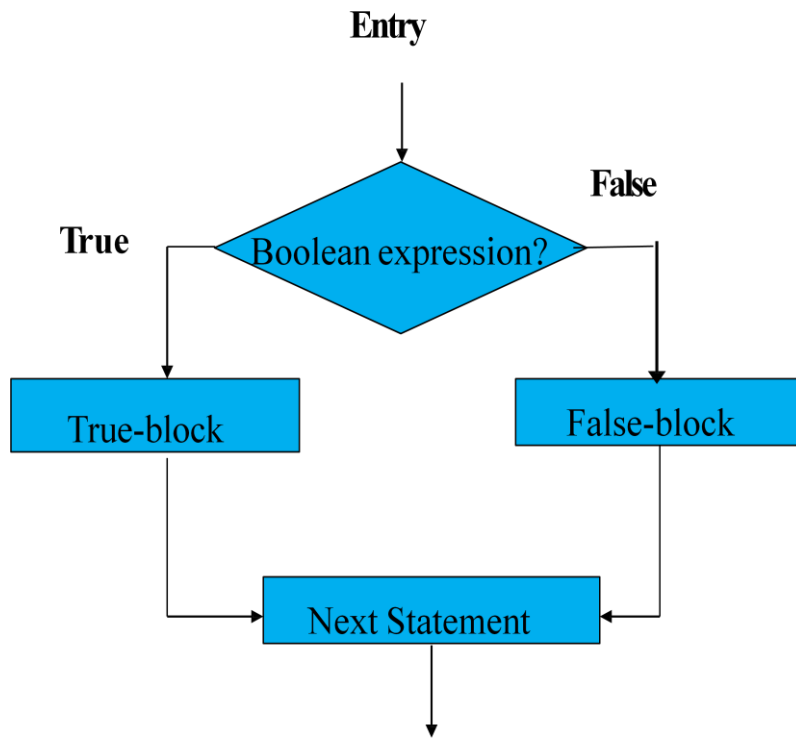
```
#include<stdio.h>
void main()
{
    int n=5;
    if(n<25)
    {
        printf("This is if statement");
    }
}
```

Output:
This is if statement

if .. else statement

Syntax

```
if(boolean expression)
{
    True-block statements;
}
else
{
    False-block statements;
}
Next statement;
```



```
#include<stdio.h>
void main()
{
int age;
printf("Enter the age");
scanf("%d",&age);
if(age>18)
{
printf("Eligible to vote");
}
else
{
printf("Not eligible to vote");
}
}
```


Cascading if..else

Syntax:

```
if (condition1)
```

```
{
```

```
statement-1
```

```
}
```

```
....
```

```
else if(condition-n)
```

```
{
```

```
statement-n
```

```
}
```

```
Else
```

```
{
```

```
default statement
```

```
}
```

```
next statement
```

```
//program to find largest three number
```

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

```
void main()
```

```
{
```

```
int n1,n2,n3;
```

```
printf("Enter the number");
```

```
scanf("%d%d%d",&n1,&n2,&n3);
```

```
if(n1>n2 && n1>n3)
```

```
{
```

```
printf("%d is largest number",n1);
```

```
}
```

```
else If(n2>n3)
```

```
{
```

```
printf("%d is the largest number",n2);
```

```
}
```

```
else
```

```
{
```

```
printf("%d is the largest number",n3);
```

```
}
```

```
}
```

Switch Statement

The switch-case conditional construct is a more structured way of testing for multiple conditions rather than resorting to a multiple if statement

Syntax:

```
switch(expression)
{
case 1: case 1 block
break;
case 2: case 2 block
break;
default: default block;
break;
}
statement;
```

/* This is an example of a switch case statement*/

```
#include<stdio.h>
Void main()
{
int w;
printf("Enter the week");
scanf("%d",&w);
switch(w)
{
case 1:
printf("Sunday");
break;
case 2:
printf("Monday");
break;
case 3:
printf("Tuesday");
break;
case 4:
printf("Wednesday");
break;
case 5:
printf("Thursday");
break;
case 6:
printf("Friday");
break;
```

```
case 7:
printf("Saturday");
break;
Default:
Printf("Invalid input please enter number between (1 – 7)");
}
}
```

Looping Statement

A loop execute the sequence of statements many times until the stated condition becomes false.

Looping statements are

- for
- while
- do while

for Loop

The for loop initialize the value before the first step. Then checking the condition against the current value of variable and execute the loop statement and then perform the step taken for each execution of loop body. For-loops are also typically used when the number of iterations is known before entering the loop.

Syntax

```
for(initialization; condition; increment/decrement)
{
Body of the loop
}
```

```
/* This is an example of a for loop */
```

```
#include<stdio.h>
void main()
{
int i;
for(i=0;i<=5;i++)
{
printf("i:",i);
}
```

Output:

```
i: 1
i: 2
i: 3
i: 4
i: 5
```

While Loop

It's a **entry controlled loop**, the condition in the while loop is evaluated, and if the condition is true, the code within the block is executed. This repeats until the condition becomes false

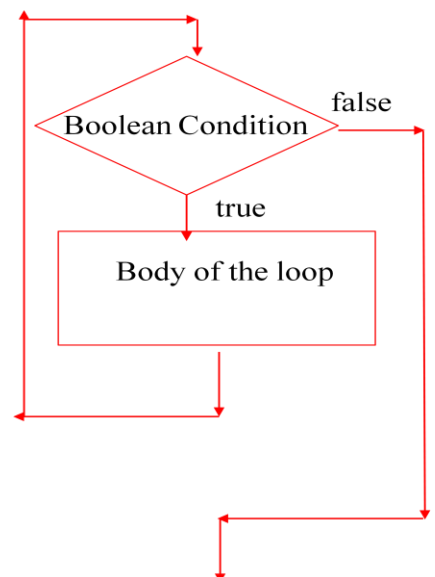
```
Syntax
while(condition)
{
Body of the loop
}
```

```
/* This is an example for a while loop */
```

```
#include<stdio.h>
void main()
{
int i = 0;
while (i < 5)
{
printf("i: ",i);
i = i + 1;
}
```

Output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
```



do.. while Loop

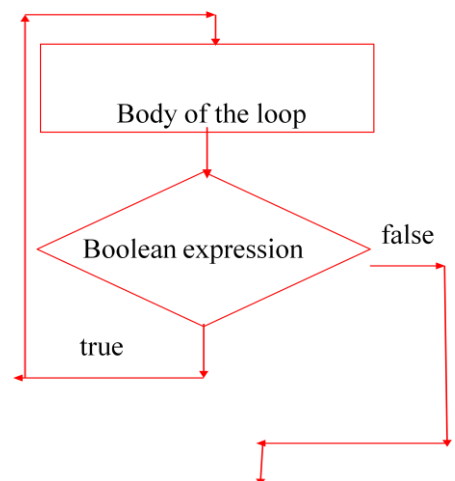
It's a **exit controlled loop**, the body of the loop gets executed first followed by checking the condition. Continues with the body if the condition is true, else loops gets terminated.

Syntax

```
do
{
body of the loop
}
while(Boolean expression);
```

/* This is an example of a do-while loop */

```
#include<stdio.h>
void main()
{
int i=5;
do
{
println("i: ",i);
i = i + 1;
}
while (i < 5);
}
```



Output:

i: 5

Pre-processor Directives

This preprocessor is a macro processor this is used automatically by the C compiler to transform your program before actual compilation. It is called macro processor because it allows you to define macros, which are brief abbreviations of longer constructs. A macro is a segment of code which is replaced by the value of macro. Macro is defined by **#define** directive.

Preprocessing directives are lines in your program that start with #. The # is followed by an identifier that is the directive name. For example, #define is the directive that defines a macro. Whitespace is also allowed before and after the #.

The # and the directive name cannot come from a macro expansion. For example, if foo is defined as a macro expanding to define, that does not make #foo a valid preprocessing directive.

Some of the preprocessor directives are:

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif
- #error
- #pragma

#include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files.

#define

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.

Syntax:

#define token value

#undef

To undefine a macro means to cancel its definition. This is done with the #undef directive.

Syntax:

#undef token

```
#include<stdio.h>
#define PI 3.1415
#undef PI
Main()
{
Printf("%f",PI);
}
```

#ifdef

The `#ifdef` preprocessor directive checks if macro is defined by `#define`. If yes, it executes the code.

Syntax:

```
#ifdef MACRO
//code
#endif
```

#ifndef

The `#ifndef` preprocessor directive checks if macro is not defined by `#define`. If yes, it executes the code.

Syntax:

```
#ifndef MACRO
//code
#endif
```

#if

The `#if` preprocessor directive evaluates the expression or condition. If condition is true, it executes the code

Syntax:

```
#if expression
//code
#endif
```

#else

The `#else` preprocessor directive evaluates the expression or condition if condition of `#if` is false. It can be used with `#if`, `#elif`, `#ifdef` and `#ifndef` directives.

Syntax:

```
#if  
//code  
#else  
//else code  
#endif
```

#error

The `#error` preprocessor directive indicates error. The compiler gives fatal error if `#error` directive is found and skips further compilation process.

```
#include<stdio.h>  
#ifndef _MATH_  
#error First include then compile  
#else  
void main()  
{  
int a;  
a=sqrt(9);  
printf("%f",a);  
}  
#endif
```

#pragma

The `#pragma` preprocessor directive is used to provide additional information to the compiler. The `#pragma` directive is used by the compiler to offer machine or operating-system feature. Different compilers can provide different usage of `#pragma` directive.

Syntax:

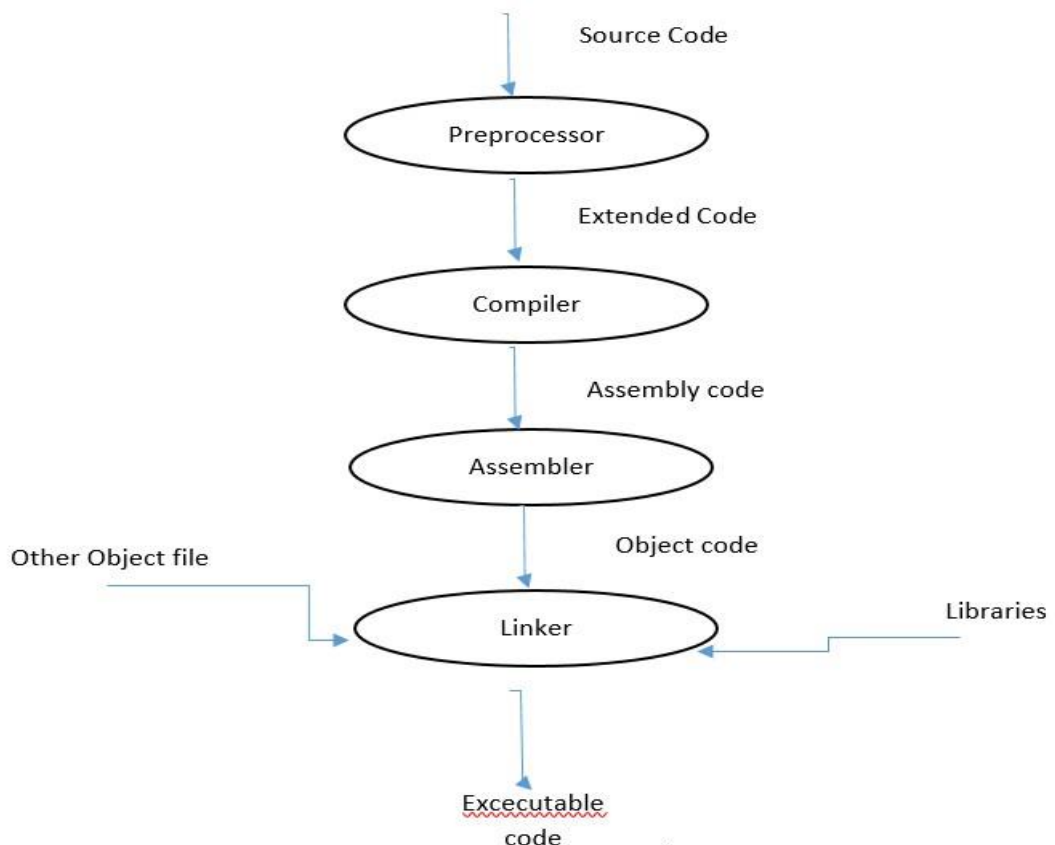
```
#pragma token
```


Compilation Process:

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.



The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.



Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is '.o'. If the name of the source file is '**welcome.c**', then the name of the object file would be 'hello.obj'.

Linker

Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'. For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.

Exercise:

Check whether the required amount can be withdrawn based on the available amount

```
#include <stdio.h>
unsigned long amount=1000, deposit, withdraw;
int ch, pin, l;
char transaction ='y';
void main()
{
while (pin != 9090)
{
printf("ENTER YOUR SECRET PIN NUMBER:");
scanf("%d", &pin);
if (pin != 9090)
printf("PLEASE ENTER VALID PASSWORD\n");
}
do
{
printf("*****Welcome to ATM Service*****\n");
printf("1. Check Balance\n");
printf("2. Withdraw Cash\n");
printf("3. Deposit Cash\n");
printf("4. Quit\n");
printf("*****?*\n\n");
printf("Enter your choice: ");
scanf("%d", &ch);
switch (ch)
{
case 1:
printf("\n YOUR BALANCE IN Rs : %lu ", amount);
break;
case 2:
printf("\n ENTER THE AMOUNT TO WITHDRAW: ");
scanf("%lu", &withdraw);
if (withdraw % 100 != 0)
{
printf("\n PLEASE ENTER THE AMOUNT IN MULTIPLES OF 100");
}
else if (withdraw >(amount - 500))
{
printf("\n INSUFFICIENT BALANCE");
}
}
```

```

else
{
amount = amount - withdraw;
printf("\n\n PLEASE COLLECT CASH");
printf("\n YOUR CURRENT BALANCE IS%lu", amount);
}
break;
case 3:
printf("\n ENTER THE AMOUNT TO DEPOSIT");
scanf("%lu", &deposit);
amount = amount + deposit;
printf("YOUR BALANCE IS %lu", amount);
break;
case 4:
printf("\n THANK U USING ATM");
break;
default:
printf("\n INVALID CHOICE");
}
printf("\n\n DO U WANT TO CONTINUE?(y/n): \n");
flush(stdin);
scanf("%c", &transaction);
if (transaction == 'n' || transaction == 'N')
l = 1;
}
while (!l);
printf("\n\n THANKS FOR USING OUT ATM SERVICE");
}

```

Menu – driven program to find the area of different shape

```

#include <stdio.h>
void main ()
{
    int ch,rad,length,width,breadth,height;
    float area;
    printf("Input 1 for area of circle\n");
    printf("Input 2 for area of rectangle\n");
    printf("Input 3 for area of triangle\n");
    printf("Input your choice : ");
    scanf("%d",&ch);

```

```

switch(ch)
{
case 1:
printf("Input radius of the circle : ");
scanf("%d",&rad);
area=3.14*rad*rad;
break;
case 2:
printf("Input length and width of the rectangle : ");
scanf("%d%d",&length,&width);
area=length*width;
break;
case 3:
printf("Input the base and height of the triangle :");
scanf("%d%d",&breadth,&height);
area=.5*breadth*height;
break;
}
printf("The area is : %f\n",area);
}

```

Find the sum of even numbers

```

#include <stdio.h>
void main()
{
    int i, x, sum=0;
    /* Input upper limit from user */
    printf("Enter upper limit: ");
    scanf("%d", &x);
    for(i=2; i<=x; i+=2)
    {
        /* Add current even number to sum */
        sum = sum+i;
    }
    printf("Sum of all even number between 1 to %d = %d", x, sum);
}

```

Assignment

Unit I

Assignment Questions

CO 1	Develop C program solutions to simple computational problems		
1.	<p>Write a C program to check whether the number is palindrome number or not a palindrome number.</p> <p>Test Data :</p> <p>Input a three digit number : 121</p> <p>Expected Output :</p> <p>The given number is : 121</p> <p>The given number is palindrome number</p>	K2	CO1
2.	<p>Write a program in C to calculate factorial of a given number.</p> <p>Test Data :</p> <p>Input the given number :5</p> <p>Expected Output :</p> <p>The factorial of a given number is : 120</p>	K2	CO1

Part A

Question & Answer

PART A QUESTION & ANSWERS

1. What is Token? (CO1)(K2)

Token is a building block of a program. A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol.

2. What is Keyword? (CO1)(K2)

Keywords are special reserved words associated with some meaning.

3. What is keyword auto for? (CO1)(K2)

By default, every local variable of the function is automatic (auto). In the below function both the variables „x“ and „y“ are automatic variables.

```
void fun()
{
    int x,
    auto int q;
}
```

4. What are main characteristics of C language? (CO1)(K2)

C is a procedural language. The main features of C language include low-level access to memory, simple set of keywords, and clean style. These features make it suitable for system programming like operating system or compiler development.

5. What are reserved words? (CO1)(K2)

Reserved words are words that are part of the standard C language library. This means that reserved words have special meaning and therefore cannot be used for purposes other than what it is originally intended for. Examples of reserved words are float, default, and return.

6. What are the types of C tokens? (CO1)(K2)

C tokens are of six types. They are,

Keywords	(eg: int, while),
Identifiers	(eg: main, total),
Constants	(eg: 10, 20),
Strings	(eg: "total", "hello"),
Special symbols	(eg: (), {}),
Operators	(eg: +, /, -, *)

7. What is the use of printf() and scanf()? (CO1)(K2)

printf(): The printf() function is used to print the integer, character, float and string values on to the screen.

Following are the format specifier:

- **%d:** It is a format specifier used to print an integer value.
- **%s:** It is a format specifier used to print a string.
- **%c:** It is a format specifier used to display a character value.
- **%f:** It is a format specifier used to display a floating point value.

scanf(): The scanf() function is used to take input from the user.

8. What is data types in C? (CO1)(K2)

- Data types in C language are defined as the data storage format that a variable can store a data to perform a specific operation.
- Data types are used to define a variable before to use in a program.
- Size of variable, constant and array are determined by data types.

9. What is typecasting? (CO1)(K2)

The typecasting is a process of converting one data type into another is known as typecasting. If we want to store the floating type value to an int type, then we will convert the data type into another data type explicitly.

(type-name) expression

10. What is the difference between variable declaration and variable definition?

(CO1)(K2)

Declaration associates type to the variable whereas definition gives the value to the variable.

11. What are global variable and how do you declare them? (CO1)(K1)

Global variables are variables that can be accessed and manipulated anywhere in the program. To make a variable global, place the variable declaration on the upper portion of the program, just after the pre_processor directives section.

12. What is local variable in C (CO1)(K2)

- The variables which are having scope/life only within the function are called local variables.
- These variables are declared within the function and can't be accessed outside the function.

13. What is constant in C (CO1)(K2)

- Constants refer to fixed values. They are also called as literals.
- C Constants are also like normal variables. But, only difference is, constant values can't be modified by the program once they are defined. Constants may be belonging to any of the data type.

14. What are the types of constants in C? (CO1)(K2)

- Integer constants
- Real or Floating point constants
- Octal & Hexadecimal constants
- Character constants
- String constants
- Backslash character constants

15. What is the difference between = and == symbol? (CO1)(K2)

The = symbol is often used in mathematical operations. It is used to assign a value to a given variable. On the other hand, the == symbol, also known as "equal to" or "equivalent to", is a relational operator that is used to compare two values.

16. Describe the order of precedence with regards of operator in C. (CO1)(K2)

Order of precedence determines which operation must first take place in an operation statement or conditional statement. On the top most level of precedence are the unary operators !, +, - and &. It is followed by the regular mathematical operators (*, / and modulus % first, followed by + and -). Next in line are the relational operators <, <=, >= and >. This is then followed by the two equality operators == and !=. The logical operators && and || are next evaluated. On the last level is the assignment operator =.

17. What is the difference between pre-increment operator and post increment operator? **(CO1)(K2)**

- Pre increment operator is used to increment variable value by 1 before assigning the value to the variable.
- Post increment operator is used to increment variable value by 1 after assigning the value to the variable.

18. What are all decision control statement in C? **(CO1)(K2)**

There are 3 types of decision making control statements in C language. They are,

1. if statements
2. if else statements
3. nested if statements

19. What will happen if break statement is not used in switch case in C? **(CO1)(K2)**

- Switch case statements are used to execute only specific case statements based on the switch expression.
- If we do not use break statement at the end of each case, program will execute all consecutive case statements until it finds next break statement or till the end of switch case block.

20. What is nested loop? **(CO1)(K2)**

A nested loop is a loop that runs within another loop. Put it in another sense, you have an inner loop that is inside an outer loop. In this scenario, the inner loop is performed a number of times as specified by the outer loop. For each turn on the outer loop, the inner loop is first performed.

21. What is the difference between while and do...while loop in C? **(CO1)(K2)**

- While loop is executed only when given condition is true.
- Whereas, do-while loop is executed for first time irrespective of the condition. After executing while loop for first time, then condition is checked.

Part B

Questions

PART B QUESTION BANK

1. Explain in detail about Datatypes in C. **(CO1)(K2)**
2. Explain about structure of a C program. **(CO1)(K2)**
3. Explain in detail about Arithmetic and Relational operators in C. **(CO1)(K2)**
4. Explain in detail about Bitwise and logical operators in C. **(CO1)(K2)**
5. Explain in detail about operator precedence. **(CO1)(K2)**
6. Illustrate about Conditional statement with example program. **(CO1)(K2)**
7. Describe iterative statement with example program. **(CO1)(K1)**
8. Write a program using for loop to calculate factorial of a number. **(CO1)(K1)**
9. Write a program to check whether the given number is Armstrong or not.
(CO1)(K3)
10. Write a program to find the sum of a digit. **(CO1)(K1)**
11. Write a program to find the given number is positive or negative or zero.
(CO1)(K2)
12. Write a program if the word is "Programming in C" display it in reverse manner. **(CO1)(K1)**
13. Write a program to display the month in order using switch case. **(CO1)(K2)**
14. Write a program to print the following pattern **(CO1)(K1)**.

*

**

15. Write a program to check whether the required amount can be withdrawn based on the available amount. **(CO1)(K1)**
16. Write a menu driven program to find the area of different shapes.
(CO1)(K2)
17. Write a program to find the sum of even number. **(CO1)(K3)**

Supportive Online Certification

Unit I

Certification Courses

⚙ NPTEL

Problem solving through Programming in C

<https://nptel.ac.in/courses/106/105/106105171/>

⚙ Coursera

1) C for Everyone: Structured Programming

<https://www.coursera.org/learn/c-structured-programming>

2) C for Everyone: Programming Fundamentals

<https://www.coursera.org/learn/c-for-everyone>

Real time Applications

Unit I

Real-Time implementation in C programming

- 1. Operating Systems**
- 2. Development of New Language**
- 3. Computation Platforms**
- 4. Embedded Systems**
- 5. Graphics and Games**

Content beyond syllabus

Unit I

Content beyond syllabus

Problem Solving and Algorithms

Learn a basic process for developing a solution to a problem. This process can be used to solve a wide variety of problems.

An algorithm development process

- ✿ **Obtain a description of the problem.**
- ✿ **Analyze the problem**
- ✿ **Develop a high-level algorithm**
- ✿ **Refine the algorithm by adding more details**
- ✿ **Review the algorithm.**

Assessment Schedule

Unit I

Prescribed Text book & References

Unit I

Text books & References

TEXT BOOK

1. Reema Thareja, "Programming in C", Oxford University Press, Second Edition, 2016

REFERENCES:

1. Kernighan, B.W and Ritchie, D. M, "The C Programming language", Second Edition, Pearson Education, 2006
2. Paul Deitel and Harvey Deitel, "C How to Program", Seventh edition, Pearson Publication
3. Juneja, B. L and Anita Seth, "Programming in C", CENGAGE Learning India pvt. Ltd., 2011
4. Pradip Dey, Manas Ghosh, "Fundamentals of Computing and Programming in C", First Edition, Oxford University Press, 2009

Mini Project Suggestions

Unit I

Mini_Projects in C Language

- 1. Cricket Score Board Project**
- 2. Customer Billing System**
- 3. Hospital Management System**
- 4. Calendar Application**
- 5. Medical Store Management System**

Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

OCS752

INTRODUCTION TO C

PROGRAMMING

Department: : Electrical and Electronics Engineering

Batch/Year: 2017-2021

Created by: Dr. S. Meenakshi

Date: 13-07-2020

Table of Contents

- ✿ Course Objectives
- ✿ Syllabus
- ✿ Course Outcomes (Cos)
- ✿ CO-PO Mapping
- ✿ Lecture Plan
- ✿ Activity based learning
- ✿ Lecture notes
- ✿ Assignments
- ✿ Part A Q&A
- ✿ Part B Qs
- ✿ List of Supportive online Certification courses
- ✿ Real time applications in day to day life and to industry
- ✿ Contents beyond Syllabus
- ✿ Assessment Schedule (proposed and actual date)
- ✿ Prescribed Text Books & Reference Books
- ✿ Mini Project Suggestions

Course Objectives

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C

3 0 0 3

OBJECTIVES

- ✿ To develop C Programs using basic programming constructs
- ✿ To develop C programs using arrays and strings
- ✿ To develop applications in C using functions and structures

Syllabus

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C 3 0 0 3

UNIT I INTRODUCTION

9

Structure of C program – Basics: Data Types – Constants – Variables – Keywords – Operators: Precedence and Associativity – Expressions – Input/output statements, Assignment statements – Decision-making statements – Switch statement – Looping statements – Pre-processor directives – Compilation process – Exercise Programs: Check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers

UNIT II ARRAYS

9

Introduction to Arrays – One dimensional arrays: Declaration – Initialization – Accessing elements – Operations: Traversal, Insertion, Deletion, Searching – Two dimensional arrays: Declaration – Initialization – Accessing elements – Operations: Read – Print – Sum – Transpose – Exercise Programs: Print the number of positive and negative values present in the array – Sort the numbers using bubble sort – Find whether the given matrix is diagonal or not.

UNIT III STRINGS

9

Introduction to Strings – Reading and writing a string – String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse – Substring – Insertion – Indexing – Deletion – Replacement – Array of strings – Introduction to Pointers – Pointer operators – Pointer arithmetic – Exercise programs: To find the frequency of a character in a string – To find the number of vowels, consonants and white spaces in a given text – Sorting the names.

UNIT IV FUNCTIONS

9

Introduction to Functions – Types: User-defined and built-in functions – Function prototype – Function definition – Function call – Parameter passing: Pass by value – Pass by reference – Built-in functions (string functions) – Recursive functions – Exercise programs: Calculate the total amount of power consumed by 'n' devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)

UNIT V STRUCTURES

9

Introduction to structures – Declaration – Initialization – Accessing the members – Nested Structures – Array of Structures – Structures and functions – Passing an entire structure – Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

TOTAL:45 PERIODS

Course Outcomes

- ✿ CO 1 - Develop algorithmic solutions to simple computational problems K1
- ✿ CO 2 - Read, write, execute by hand simple C programs. K2
- ✿ CO 3 - Structure simple C programs for solving problems using statements K2
- ✿ CO 4 – Represent data using arrays and strings operations K3
- ✿ CO 5 - Decompose a C program into functions and pointers K3
- ✿ CO 6 - Represent and write program using structure and union K3

CO – PO Mapping

CO	PO	Mapping Level	Justification
CO	PO	Mapping Level	Justification
CO1	PO1	2	Apply simple mathematical concepts for writing algorithms
CO1	PO2	2	Identify formulae for the given problem
CO1	PO3	2	Design algorithmic way of problem solving
CO1	PO5	2	Recognize the need of algorithm in implementation
CO1	PO12	1	Apply logic to solve simple problem statement
CO2	PO1	3	Identify the data type and operators to solve the problem
CO2	PO2	3	Design the expression in an efficient way
CO2	PO3	2	Recognize the need of basic data types and operators
CO2	PO5	2	Apply control flow statement for solving the problem
CO2	PO12	1	Formulate the algorithm into executable c code
CO3	PO1	3	Develop a complete program in a simple way
CO3	PO2	3	Recognize the need of control flow statements
CO3	PO3	3	Apply the knowledge to find the possible code for function
CO3	PO5	2	Identify the code for decomposition as function
CO3	PO12	1	Develop functions and reuse it whenever required to reduce the lines of code
CO4	PO1	2	Recognize the need of function concepts
CO4	PO2	2	Apply compound data knowledge to select any one
CO4	PO3	2	Apply the concept of pointers
CO4	PO5	2	Design and Develop program using the selected compound data
CO5	PO12	1	Recognize the need of structure
CO5	PO1	2	Apply the basic idea of handling with union
CO6	PO2	2	Identify the number of modes and operations on structure in detail
CO6	PO3	2	Develop programs using structure and union

Lecture Plan

Unit II

Unit II - Arrays in C

S.No	Topics	No. of Periods	Proposed Date	Actual Lecture Date	Pertaining CO	Taxonomy Level	Mode of Delivery
1	Introduction to Arrays – One dimensional arrays: Declaration – Initialization -	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
2	Accessing elements Operations: Traversal, Selection,	1			CO1	K2	PPT, Chalk & Talk
3	Insertion, Deletion, Searching	1			CO1	K2	PPT, Chalk & Talk
4	Two dimensional arrays: Declaration – Initialization -	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
5.6	Accessing elements, Operations: Read – Print – Sum – Transpose	2			CO1	K2	PPT, Chalk & Talk
7	Sorting operations	1			CO1	K2	PPT, Chalk & Talk
8,9	Exercise Programs: Ex. Prog. 1 : Print the number of positive and negative values present in the array – Ex. Prog. 2 :Sort the numbers using bubble sort – Ex. Prog. 3 : Find whether the given is matrix is diagonal or not. and industrial case studies	2			CO1	K2	PPT, Chalk & Talk

Activity Based Learning

Unit II

Activity Based Learning

- ✿ Learn by solving problems – Tutorial Sessions can be conducted
 - Tutorial sessions available in Skillrack for practice
- ✿ Learn by questioning
- ✿ Learn by doing hands-on IN ONLINR / VIRTUAL LAB.

Lecture Notes

UNIT II ARRAYS

9

Introduction to Arrays – One dimensional arrays: Declaration – Initialization - Accessing elements –Operations: Traversal, Insertion, Deletion, Searching - Two dimensional arrays: Declaration –Initialization - Accessing elements – Operations: Read – Print – Sum – Transpose – Exercise Programs: Print the number of positive and negative values present in the array – Sort the numbers using bubble sort - Find whether the given matrix is diagonal or not.

Unit II - Arrays in C : LEARNING PLAN

Topic 2.0 and 2.1

Sl. No.	Topics	Learning Content (hh.mm)	Post-Session (Quiz + Assignment) (hh:mm)
2.0 and 2.1	Introduction to Arrays – One dimensional arrays: Declaration – Initialization - Accessing elements Operations: Traversal, Insertion, Deletion, Searching	3.00	1.00
2.2	Two dimensional arrays: Declaration –Initialization - Accessing elements, Operations: Read – Print – Sum – Transpose, Sorting	3.00	0.50
2.3	Exercise Programs: Ex. Prog 1 : Print the number of positive and negative values present in the array – Ex. Prog 2 :Sort the numbers using bubble sort - Ex. Prog 3 : Find whether the given is matrix is diagonal or not. and More programs on C, Industrial case studies	3.00	1.00
	Total	9.00	2.50

Introduction to Arrays

Topic. 2.0

Introduction to Array

Array Intro.

An array is a collection of similar data elements.

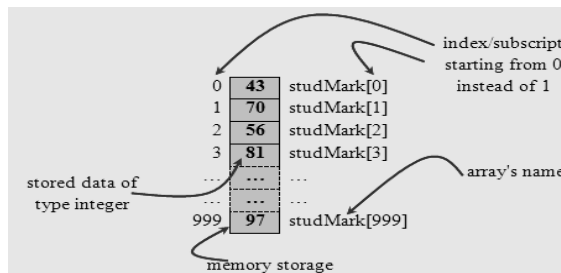
These data elements have the same data type.

The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript). If one subscript, then we call as one dimensional array.

Memory representation in an array

The array elements are stored in contiguous memory locations.

For the array, `int stuMark[]={43,70,56}`; the memory representation shown as follows:



By using an array, we just declare like this,

```
int studMark[1000];
```

This will reserve 1000 contiguous memory locations for storing the students' marks. Graphically, this can be depicted as in the following figure.

Compared to the basic data type (int, float, char and double) it is an aggregate or derived data type.

All the elements of an array occupy a set of contiguous memory locations.
Why need to use array type?

Consider the following issue:

"We have a list of 1000 students' marks of an integer type. If using the basic data type (int), we will declare something like the following..."

```
int studMark0, studMark1, studMark2, ..., studMark999
```

Can you imagine how long we have to write the declaration part by using normal variable declaration?

```
int main(void){  
    int studMark1, studMark2, studMark3, studMark4, ..., ...,  
    studMark998, stuMark999, studMark1000;  
    ...  
    ...  
    return 0;}
```

This absolutely has simplified our declaration of the variables. We can use index or subscript to identify each element or location in the memory.

Hence, if we have an index of jIndex, studMark[jIndex] would refer to the jIndexth element in the array of studMark.

For example, studMark[0] will refer to the first element of the array.

Thus by changing the value of jIndex, we could refer to any element in the array.

So, array has simplified our declaration and of course, manipulation of the data.

One Dimensional Arrays

Topic 2.1

Declaration

Initialization

Accessing elements

Operations

Traversal, (*+selection)

Insertion,

Deletion,

Searching

*** TO BE COVERED FOR THE TOPICS**

One Dimensional Array

One/Single Dimensional array

Dimension refers to the array's size, which is how big the array is.

Declaration of One Dimensional array

Declaring an 1D dimnl. array means specifying three things:

The data type- what kind of values it can store ex, int, char, float

Name- to identify name of the array

The size- the maximum number of values that the array can hold

Arrays are declared using the following syntax.

```
type name[size];
```

For example, to declare an array of 30 characters, that construct a people name, we could declare,

```
char cName[30];
```

Which can be depicted as follows,

In this statement, the array character can store up to 30 characters with the first character occupying location cName[0] and the last character occupying cName[29].

Note that the index runs from 0 to 29. In C, an index always starts from 0 and ends with array's (size-1).

So, take note the difference between the array size and subscript/index terms.

J	cName[0]
o	cName[1]
d	cName[2]
i	cName[3]
e	cName[4]
...	cName[5]
...	...
...	...
r	cName[29]

Examples of the one-dimensional array declarations,

```
int    xNum[20], yNum[50];
float  fPrice[10], fYield;
char   chLetter[70];
```

The first example declares two arrays named xNum and yNum of type int. Array xNum can store up to 20 integer numbers while yNum can store up to 50 numbers. The second line declares the array fPrice of type float. It can store up to 10 floating-point values, fYield is basic variable which shows array type can be declared together with basic type provided the type is similar. The third line declares the array chLetter of type char. It can store a string up to 69 characters.

Note: Why 69 instead of 70? Remember, a string has a null terminating character (\0) at the end, so we must reserve for it.

Initialization of an array

An array may be initialized at the time of declaration.

Giving initial values to an array.

Initialization of an array may take the following form,

```
type  array_name[size] = {a_list_of_value};
```

For example:

```
int    idNum[7] = {1, 2, 3, 4, 5, 6, 7};
float  fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
char   chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
```

The first line declares an integer array idNum and it immediately assigns the values 1, 2, 3, ..., 7 to idNum[0], idNum[1], idNum[2],..., idNum[6] respectively.

The second line assigns the values 5.6 to fFloatNum[0], 5.7 to fFloatNum[1], and so on.

Similarly the third line assigns the characters 'a' to chVowel[0], 'e' to chVowel[1], and so on.

Note: again, for characters we must use the single apostrophe/quote (') to enclose them.

Also, the last character in chVowel is NULL character ('\0').

Initialization of an array of type char for holding strings may take the following form,

```
char array_name[size] = "string_lateral_constant";
```

For example, the array chVowel in the previous example could have been written more compactly as follows,

```
char chVowel[6] = "aeiou";
```

When the value assigned to a character array is a string (which must be enclosed in double quotes), the compiler automatically supplies the NULL character but we still have to reserve one extra place for the NULL.

For unsized array (variable sized), we can declare as follow,

```
char chName[ ] = "Mr. Dracula";
```

C compiler automatically creates an array which is big enough to hold all the initializer.

Store values in the array (3 possible ways) →

1) Initialize the elements

2) Inputting Values for the elements

3) Assigning Values to the elements

1) Initialize the elements

```
int idNum[7] = {1, 2, 3, 4, 5, 6, 7};  
float fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};  
char chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
```

2) Inputting Values for the elements

```
int i, marks[10];  
for(i=0;i<10;i++)  
    scanf("%d", &marks[i]);
```

3) Assigning Values to the elements

```
int i, arr1[10], arr2[10];  
for(i=0;i<10;i++)  
    arr2[i] = arr1[i];
```


Accessing elements

To access all the elements of the array, you must use a loop. That is, we can access all the elements of the array by varying the value of the subscript into the array. But note that the subscript must be an integral value or an expression that evaluates to an integral value.

```
int i, marks[10];  
for(i=0;i<10;i++)  
    marks[i] = -1;
```

Calculating the address of array elements

Address of data element, $A[k] = BA(A) + w(k - \text{lower_bound})$

Here, A is the array

k is the index of the element of which we have to calculate the address

BA is the base address of the array A.

w is the word size of one element in memory, for example, size of int is 2.

99	67	78	56	88	90	34	85
Marks[0] 1000	marks[1] 1002	marks[2] 1004	marks[3] 1006	marks[4] 1008	marks[5] 1010	marks[6] 1012	marks[7] 1014

$$\begin{aligned} \text{Marks}[4] &= 1000 + 2(4 - 0) \\ &= 1000 + 2(4) = 1008 \end{aligned}$$

Calculating the length of the array

$\text{Length} = \text{upper_bound} - \text{lower_bound} + 1$

Where, upper_bound is the index of the last element

and lower_bound is the index of the first element in the array

99	67	78	56	88	90	34	85
Marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]]

Here, lower_bound = 0, upper_bound = 7

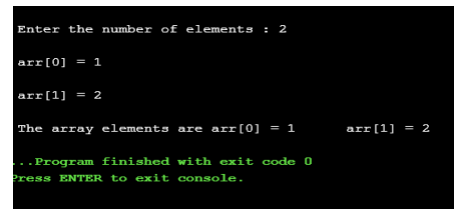
Therefore, length = $7 - 0 + 1 = 8$

Program example 1: Write A program to read and display n numbers using an array

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i=0, n, arr[20];
    printf("\n Enter the number of elements : ");
    scanf("%d", &n);

    for(i=0;i<n;i++)
    {
        printf("\n arr[%d] = ", i);
        scanf("%d",&arr[i]);
    }
    printf("\n The array elements are ");
    for(i=0;i<n;i++)
        printf("arr[%d] = %d\t", i, arr[i]);
    return 0;
}
```

OUTPUT:

A screenshot of a terminal window showing the execution of the first program. The user enters '2' for the number of elements. The program then prompts for two array elements, '1' and '2'. It displays the array elements as 'arr[0] = 1' and 'arr[1] = 2'. Finally, it prints 'The array elements are arr[0] = 1 arr[1] = 2' and ends with '...Program finished with exit code 0 Press ENTER to exit console.'

Arrays allow programmers to group related items of the same data type in one variable. However, when referring to an array, one has to specify not only the array or variable name but also the index number of interest.

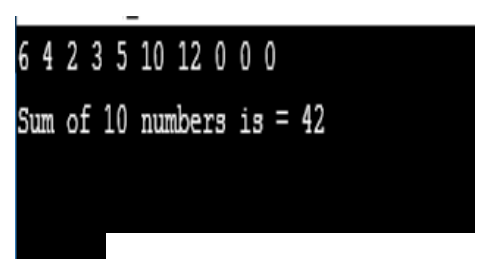
Program example 2: Sum of array's elements

```
// finding sum of array's element
#include <stdio.h>
// replace every nSize occurrences with 10
#define nSize 10

int main(void){
    int iCount, nSum = 0, iNum[nSize] = {6,4,2,3,5,10,12};

    f.or (iCount=0; iCount<nSize; iCount++)    {
        // display the array contents
        printf("%d ",iNum[iCount]);
        // do the summing up
        nSum = nSum + iNum[iCount];
    }
    // display the sum
    printf("\nSum of %d numbers is = %d\n", iCount, nSum);
    return 0;
}
```

OUTPUT:

A screenshot of a terminal window showing the execution of the second program. It displays the array elements '6 4 2 3 5 10 12 0 0 0' and then prints 'Sum of 10 numbers is = 42'.

Note: the array's element which is not initialized is set to 0 automatically

GUIDED ACTIVITY1 – Arrays

i) C has **no bounds checking** on arrays. One could overwrite either end of an array and write into some other variable's data or even into the program's code.

E.g.

```
int count[10],i;
for (i=0;i<100;i++)
count[i] = i;
```

This code will compile without error, but it is incorrect because the for loop will cause the array count to be overrun.

ii) e.g. **int aiArrayOfIntegers[10];**

That is, the ten elements of the array can be referenced as

**aiArrayOfIntegers[0],aiArrayOfIntegers[1],
aiArrayOfIntegers[2], ... , aiArrayOfIntegers[9].**

These values may be accessed as shown below.

`Var1 = aiArrayOfIntegers[0];`

Note that in the declaration statement, aiArrayOfIntegers[10] means that there are 10 elements in the array; but in an assignment statement, aiArrayOfIntegers[10] refers to the 11th element of a 10-element array. That is, if there are ten elements in an array, the **subscript of the last element of the array is 9, and not 10. This is a common programming mistake and results in some indeterminate value being returned by aiArrayOfIntegers[10],** which is very likely to cause program failure.

iii) In the **declaration statement, the subscript can be a named constant;** however, it cannot be a variable.

IV) The amount of **storage required to hold an array** is directly related to its type and size.

For a single dimensional array,

the total size in bytes is computed as:

total-bytes = sizeof(data-type) * length of array.

Operations

Operation on array includes: **1) Traversal, 2) selection, 3) Insertion, 4) Deletion 5) Searching**

1) Traversal

Traversal is an operation in which **each element of a list, stored in an array, is visited..**

The travel proceeds from the **zeroth element to the last element** of the list.

Exercise Program 1 : Traverse on the list and Print the number of positive and negative values present in the array -as <0,=0,>0)

Algorithm:

Step 1: get the elements

Step 2: visit all the elements from oth element to the last element.

Step 3. chk for element is <0 =0 and >0, if so do count of each criteria.

Step 4: count of negative, zero and positive in which travel proceeds from oth to last.

Step 5. print the count for each criteria.

```
#include <stdio.h>

void main()
{
    int list[10];
    int n;
    int i, neg=0, zero=0, pos=0;
    printf("\n enter the size of the list\n");
    scanf("%d",&n);
    printf("Enter the elements one by one");
    for(i=0;i<n;i++)
    {
        printf("\n Enter number %d number",i);
        scanf("%d", &list[i]);
    }
    for(i=0;i<n;i++)
    {
        if(list[i]<0)
            neg=neg+1;
        else
            if(list[i]==0)
                zero=zero+1;
            else
                pos=pos+1;
    }
    printf("No of Negative numbers in given list are %d", neg);
    printf("No of Zeros in given list are %d", zero);
    printf("No of Positive numbers in given list are %d", pos);
}
```

OUTPUT:

```
enter the size of the list
3
Enter the elements one by one
Enter number 0 number
1
Enter number 1 number
-2
Enter number 2 number
0
No of Negative numbers in given list are 1
No of Zeros in given list are 1
No of Positive numbers in given list are 1
```

2) Selection

An array allows selection of an element for given index.

Array is called as random access data structure.

Algorithm:

Step 1: enter size of the list

Step 2: enter the merit list one by one

Step 3: get into menu of two choice 1-query and 2. quit

Step 4: get the pos value and find the value in that pos value

Step 5. print that value

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float merit[10];
    int size,i,pos,choice;
    float percentage;
    printf("\n Enter the size of the list");
    scanf("%d", &size);
    printf("\n Enter the merit list one by one");
    for(i=0; i < size; i++)
    {
        printf("\n Enter Data:");
        scanf("%f", &merit[i]);
    }
    do
    {
        printf("\n menu");
        printf("\n Query.....1");
        printf("\n Quit..... 2");
        printf("\n Enter your choice");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                printf("\n Enter position");
                scanf("%d", &pos);
                percentage=merit[pos];
                printf("\n percentage=%4.2f", percentage);
                break;

            case 2:
                printf("\n Quitting");
                }
        printf("\n press a key to continue...:");}
    while(choice!=2);}
```

OUTPUT:

```
Enter the size of the list
3
Enter the merit list one after one
1 2 3
menu
Query.....1
Quit.....2
Enter your choice1
Enter position2
percentage=3.00press a key to continue...:menu
Query.....1
Quit.....2
Enter your choice
```

3) Insertion

Insertion is the operation that inserts an element at a given location of the list.

To insert an element at i^{th} location of the list, then all elements from the right of $i+1^{\text{th}}$ location have to be shifted one step towards right.

Algorithm:

Step 1: Set upper_bound = upper_bound + 1

Step 2: Set A[upper_bound] = VAL

Step 3; EXIT

Step 1: [INITIALIZATION] SET I = N

Step 2: Repeat Steps 3 and 4 while I >= POS

Step 3: SET A[I + 1] = A[I]

Step 4: SET I = I - 1

[End of Loop]

Step 5: SET N = N + 1

Step 6: SET A[POS] = VAL

Step 7: EXIT

```
#include <stdio.h>
int main()
{ int array[100], position, i, n, value;
printf("Enter number of elements in array\n");
scanf("%d", &n);
printf("Enter %d elements\n", n);
for (i = 0; i < n; i++)
scanf("%d", &array[i]);
printf("Enter the location where you wish to insert an element\n");
scanf("%d", &position);
printf("Enter the value to insert\n");
scanf("%d", &value);
for (i = n - 1; i >= position - 1; i--)
array[i+1] = array[i];array[position-1] = value;
printf("Resultant array is\n");
for (i = 0; i <= n; i++) printf("%d\n", array[i]);
return 0;
}
```

OUTPUT:

```
Enter number of elements in array
3
Enter 3 elements
1
2
3
Enter the location where you wish to insert an element
1
Enter the value to insert
4
Resultant array is
4
1
2
3
```

4) Deletion

Deletion is the operation that removes an element from a given location of the list. To delete an element from the i th location of the list, then all elements from the right of $i+1$ th location have to be shifted one step towards left to preserve contiguous locations in the array.

Algorithm:

Step 1: Set upper_bound = upper_bound - 1

Step 2: EXIT

Step 1: [INITIALIZATION] SET I = POS

Step 2: Repeat Steps 3 and 4 while I <= N - 1

Step 3: SET A[I] = A[I + 1]

Step 4: SET I = I + 1

[End of Loop]

Step 5: SET N = N - 1

Step 6: EXIT

```
#include <stdio.h>
int main()
{
    int array[100], position, i, n;
    printf("Enter number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d elements\n", n);
    for ( i = 0 ; i < n ; i++ )
        scanf("%d", &array[i]);
    printf("Enter the location where you wish to delete element\n");
    scanf("%d", &position);
    if ( position >= n+1 )
        printf("Deletion not possible.\n");
    else
    {
        for ( i = position - 1 ; i < n - 1 ; i++ )
            array[i] = array[i+1];
        printf("Resultant array is\n");
        for( i = 0 ; i < n - 1 ; i++ )
            printf("%d\n", array[i]);
    }
    return 0;
}
```

OUTPUT:

```
Enter number of elements in array
3
Enter 3 elements
1
2
3
Enter the location where you wish to delete element
3
Resultant array is
1
2
```

5) Searching

Search is an operation in which a given list is searched for a particular value. A list can be searched sequentially wherein the search for the data item starts from the beginning and continues till the end of the list. This method is called linear Search.. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.

linear Search

```
#include<stdio.h>
void main(){
    int numlist[20];
    int n,pos, val,i;
    printf("\n enter the size of the list");
    scanf("%d", &n);
    printf("\n Enter the elements one by one");
    for(i=0;i<n;i++){
        scanf("%d", &numlist[i]);}
    printf("\n Enter the value to be searched");
    scanf("%d", &val);

    for(i=0;i<n;i++){
        if(val== numlist[i]) {
            printf("%d is present at location %d.\n",val,i+1);
            break;    }
    if(i==n)
        printf("%d isn't present in the array.\n",val);
    }}
```

OUTPUT:

```
enter the size of the list3

Enter the elements one by one
2
3

Enter the value to be searched3
3 is present at location 3.
```


Binary Search

Binary search in C language to find an element in a sorted array. If the array isn't sorted, you must sort it using a sorting technique such as bubble sort, insertion or selection sort. If the element to search is present in the list, then we print its location. The program assumes that the input numbers are in **ascending** order.

```
#include<stdio.h>
int main(){
    int c, first, last, midd, n, search, array[100];
    printf("Enter number of elements:\n");
    scanf("%d",&n);
    printf("Enter %d integers:\n", n);
    for (c = 0; c < n; c++)
        scanf("%d",&array[c]);
    printf("Enter the value to find:\n");
    scanf("%d", &search);
    first = 0;
    last = n - 1;

    while (first <= last) {
        midd = (first+last)/2;
        if (array[midd] == search)
            break;
        else if (search < array[midd])
            last = midd - 1;
        else
            first = midd + 1;  }
    if (first > last)
        printf("Element not found");
    else
        printf("Element is at positoin %d",midd+1);}
```

ONE DIMENSIONAL ARRAYS FOR INTER FUNCTION COMMUNICATION are 1)
Passing individual elements and 2) Passing entire array

OUTPUT:

```
Enter number of elements:
3
Enter 3 integers:
5 7 10
Enter the value to find:
10
Element is at positoin 3
```



Test Yourself –2.0 & 2.1 Topics (Arrays in C - Introduction to Arrays – One dimensional arrays)

- 1) Which of the following correctly declares an array?
A. `int anarray[10];`
B. `int anarray;`
C. `anarray{10};`
D. `array anarray[10];`
 2. What is the index number of the last element of an array with 29 elements?
A. 29
B. 28
C. 0
D. Programmer-defined
 3. You need to decide on an optimal search algorithm to be used to search for a data which is to be stored in an alphabetically sorted order. Which method would you use?
a) Linear Search
b) Binary Search
c) Radix Search
d) None of the above
 4. Which of the following correctly accesses the seventh element stored in `foo`, an array with 100 elements?
A. `foo[6];`
B. `foo[7];`
C. `foo(7);`
D. `foo;`
 5. Which of the following gives the memory address of the first element in array `foo`, an array with 100 elements?
A. `foo[0];`
B. `foo;`
C. `&foo;`
D. `foo[1];`
 6. Does array bound checking happen in C language?
 7. What does the subscript of an array indicate?
 8. Consider the following declaration:
`int aiNum[]={10,20,30,40,50};`
- Assume that the starting address is 1000 as an analogy, what is the address of the fourth element of the array?
9. `Double adNumber[4];`
How many bytes are allocated for the above array declaration?
 10. `Int aiNum[4];`
Which of the following is the CORRECT statement to get the value into the index 2.
`scanf("%d",aiNum[2]);`
`scanf("%d",&aiNum[2]);`



REVIEW questions and answers – 2.0 & 2.1 Topics (Arrays in C - Introduction to Arrays – One dimensional arrays)

i) State Whether the following statement are true or false:

a) The type of all elements in an array must be the same

True

b) When an array is declared, c automatically initializes its elements to zero

True

c) Accessing an array outside its range is a compile time error

True

ii) Identify the errors, if any in each of declaration, by assumption ROW and COLUMN are declared as symbolic constants.

a) Float values [10,14];

incorrect

b) Int sum[];

correct

c) Double salary[i+ROW]

incorrect

iii) Fill in the blanks in the following statements

a) The variable used as a subscript in an array is popularly known as _____variable

index variable

b) An array can be initialized either at run time or at _____
compile time

c) Purpose of array is to _____

a collection of variables of the same type

A Summary on 1D Arrays

- Before using an array, its type and size must be declared
- The first element in the array is numbered 0, so the last element is 1 less than the size of the array
- The elements of the array are always stored in contiguous memory locations
- An array can be initialized at the same place where it is declared.

Example: `int num[6] = {2,4,12,5,45,5}`

if the array is initialized at the time of declaration, mentioning the dimension of array is optional.

Example: `double dNum[] = {12.3, 34.2, -23.4, -11.3};`

- If the array elements are not given any specific values, they are supposed to contain garbage values.
- In C there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array. This will lead to unpredictable results, to say the least, and there will be no error messages to warn the programmer.

Unit II - Arrays in C : LEARNING PLAN

Topic 2.2

Sl. No.	Topics	Learning Content (hh.mm)	Post-Session (Quiz + Assignment) (hh:mm)
2.0 and 2.1	Introduction to Arrays – One dimensional arrays: Declaration – Initialization - Accessing elements Operations: Traversal, Insertion, Deletion, Searching	3.00	1.00
2.2	Two dimensional arrays: Declaration –Initialization - Accessing elements, Operations: Read – Print – Sum – Transpose, Sorting	3.00	0.50
2.3	Exercise Programs: Ex. Prog 1 : Print the number of positive and negative values present in the array – Ex. Prog 2 :Sort the numbers using bubble sort - Ex. Prog 3 : Find whether the given is matrix is diagonal or not. and industrial case studies	3.00	1.00
	Total	9.00	2.50

Two Dimensional Arrays

Topic. 2.2

Multi and Two dimensional arrays

Declaration

Initialization

Accessing elements

Operations:

Read – Print – Sum – Transpose

(*+sorting, addition, subtraction, multiplication, determinant, sum of principal diagonal element)

* To be covered for the 2D matrix operations

Multi Dimensional Arrays (array of arrays)

Two Dimensional/2D Arrays

C looks a two dimensional array as an array of a one dimensional array. The 2-D array be visualized as a rectangular grid of rows and columns.

Declaration of two Dimensional/2D Arrays

A two dimensional array has two subscripts/indexes.

The first subscript refers to the row, and the second, to the column.

Its declaration has the following form,

```
data_type array_namer[row_size][column size];
```

For examples,

```
int xInteger[3][4];
```

```
float matrixNum[20][25];
```

The first line declares xInteger as an integer array with 3 rows and 4 columns.

Second line declares a matrixNum as a floating-point array with 20 rows and 25 columns.

Therefore, a two dimensional mXn array is an array that contains m*n data elements and each element is accessed using two subscripts, i and j where $i \leq m$ and $j \leq n$

```
int marks[3][5]
```

First
Dimension

R/C	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0					
Row 1					
Row 2					

Second
Dimension

Rows/Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	Marks[0][0]	Marks[0][1]	Marks[0][2]	Marks[0][3]	Marks[0][4]
Row 1	Marks[1][0]	Marks[1][1]	Marks[1][2]	Marks[1][3]	Marks[1][4]
Row 2	Marks[2][0]	Marks[2][1]	Marks[2][2]	Marks[2][3]	Marks[2][4]

Memory representation of a two dimensional array

There are two ways of storing a 2-D array can be stored in memory. **The first way is row major order and the second is column major order.**

The 2D array can be represented in two ways:

1. Row major order of storage
2. Column major order of storage

Row Major Order

In row-major order, array elements are stored row-wise

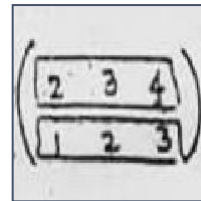
In C Language, row-major order is used

For example, i

```
int a[2][3]={{2,3,4},{1,2,3}};
```

memory representation is

2	3	4	1	2	3
3000	3002	3004	3006	3008	3010
a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]



Column Major Order

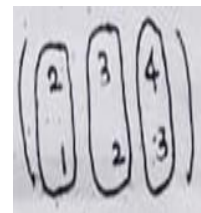
In column-major order, array elements are stored column-wise

For example, l

```
int a[3][2]={{2,1},{3,2},{4,3}};
```

memory representation is

2	1	3	2	4	3
3000	3002	3004	3006	3008	3010
a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]



Initialization of two Dimensional/2D Arrays

A two dimensional array is initialized in the same way as a single dimensional array is initialized.

The general form for initializing a 2D array is

```
datatype  
arrayname[rowsize][colsize]={{row0element},{row1element}...{rownel  
ement}}
```

For example,

```
int marks[2][3]={90, 87, 78, 68, 62, 71};  
int marks[2][3]={  
    row0    row1  
    {90,87,78},{68, 62, 71}};
```

Write a program to print the elements of a 2D array

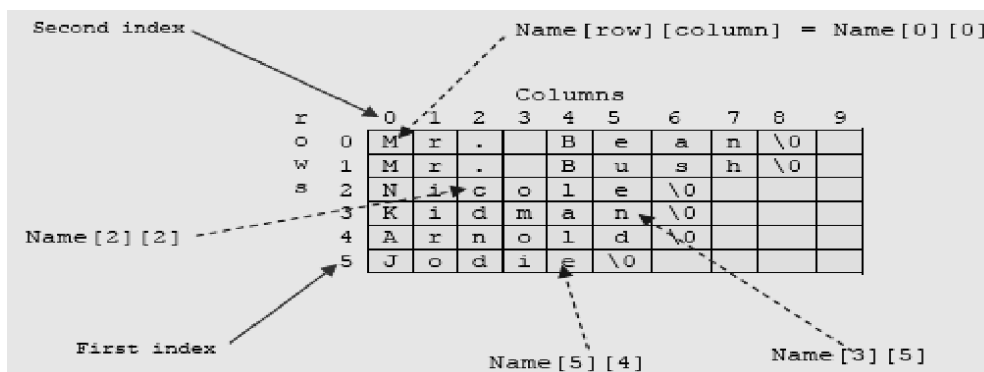
```
#include<stdio.h>  
#include<conio.h>  
main(){  
    int arr[2][2] = {12, 34, 56,32};  
    int i, j;  
    for(i=0;i<2;i++)    {  
        printf("\n");  
        for(j=0;j<2;j++)  
            printf("%d\t", arr[i][j]);  
    }  
    return 0;}
```

If we assign **initial string values for the 2D array** it will look something like the following,

```
char Name[6][10] = {"Mr. Bean", "Mr. Bush", "Nicole", "Kidman", "Arnold",  
"Jodie"};
```

Here, we can initialize the array with 6 strings, each with maximum 9 characters long.

If depicted in rows and columns it will look something like the following and can be considered as contiguous arrangement in the memory.



Take note that for strings the null character (`\0`) still needed.

From the shaded square area of the figure we can determine the size of the array. For an array `Name[6][10]`, the array size is $6 \times 10 = 60$ and equal to the number of the colored square. In general, for

```
array_name[x][y];
```

The array size is = First index **x** second index = xy .

This also true for other array dimension, for example three dimensional array,

```
array_name[x][y][z]; => First index x second index xy third index =  $xyz$ 
```

For example,

```
ThreeDimArray[2][4][7] =  $2 \times 4 \times 7 = 56$ .
```

And if you want to illustrate the 3D array, it could be a cube with wide, long and height dimensions.

Accessing of two Dimensional/2D Arrays

The element of a 2D-Array can be accessed using 2 subscripts [], a[0][0] to access the element at the 0th row and 0th column.

To access all the elements of a 2D array, use two for loops.

```
for(i=0;i<row;i++) {  
    for(j=0;j<col;j++)  
        scanf("%d", &arr[i][j]);  
    //accessing element of ith row and jth col.  
}
```

Advantages and limitations

Advantages:

Arrays support direct indexing: the time taken to access any array element the same.

Limitations:

Arrays are static: size of array cannot be expanded or squeezed at run time.

Application areas of an array

An array is an example of a static storage structure. It is used when a list of similar data needs to be stored and the number of items is known. It is frequently used in various data structure programs

GUIDED ACTIVITY 1 – 2D Arrays

The contents of the array in memory after the three strings are read in the array.

	0	1	2	3
0	y	o	u	\0
1	a	r	e	\0
2	c	a	t	\0

Re-run the program, enter the following data: "you", "my" and "lav". Illustrates the content as done previously.

	0	1	2	3
0	y	o	u	\0
1	m	y	\0	
2	l	a	v	\0

Does your output agree?

How is the null character, '\0' printed?

Is there a garbage character in a[1][3]? If so, why?

Operations on 2D array

Operations in 2D arrays include: 1) Read – 2) Print – 3) Sum – 4) Transpose etc.

1) Read – 2) Print – 3) Sum of a matrix

```
#include <stdio.h>
void main()
{
    int arr1[2][2],i,j;
    printf("\n\nRead a 2D array of size 2x2 - print the matrix and sum of the matrix:\n");
    /* Stored values into the array*/
    printf("Read elements in the matrix :\n");
    for(i=0;i<2;i++) {
        for(j=0;j<2;j++) {
            printf("element - [%d],[%d] : ",i,j);
            scanf("%d",&arr1[i][j]);
        }
    }
    printf("\nPrint the matrix is : \n");
    int sum=0;
    for(i=0;i<2;i++) {
        printf("\n");
        for(j=0;j<2;j++){
            sum=sum+arr1[i][j];
            printf("%d\t",arr1[i][j]);
        }
    }
    printf("\n\n");
    printf("Sum of the array is %d",sum);
}
```

OUTPUT

```
Read a 2D array of size 2x2 - print the matrix and sum of the matrix:
Read elements in the matrix :
element - [0],[0] : 1
element - [0],[1] : 1
element - [1],[0] : 1
element - [1],[1] : 1

Print the matrix is :

1      1
1      1

Sum of the array is 4
```

4) Transpose

Transpose of a matrix in C language: This C program prints transpose of a matrix. It is obtained by interchanging rows and columns of a matrix. For example, consider the following 3 X 2 matrix:

```
1 2
3 4
5 6
```

Transpose of the matrix:

```
1 3 5
2 4 6
```

When we transpose a matrix then its order changes, but for a square matrix, it remains the same.

```
#include <stdio.h>
void main(){
    int arr1[50][50],brr1[50][50],i,j,r,c;
    printf("\n\nTranspose of a Matrix :\n");
    printf("_____ \n");
    printf("\nInput the rows and columns of the matrix : ");
    scanf("%d %d",&r,&c);
    printf("Input elements in the first matrix :\n");
    for(i=0;i<r;i++){
        for(j=0;j<c;j++){
            printf("element - [%d],[%d] : ",i,j);
            scanf("%d",&arr1[i][j]);
        }
    }
    printf("\nThe matrix is :\n");
    for(i=0;i<r;i++){
        printf("\n");
        for(j=0;j<c;j++){
            printf("%d\t",arr1[i][j]);
        }
    }
    for(i=0;i<r;i++){
        for(j=0;j<c;j++){
            brr1[j][i]=arr1[i][j];
        }
    }
    printf("\n\nThe transpose of a matrix is : ");
    for(i=0;i<c;i++){
        printf("\n");
        for(j=0;j<r;j++){
            printf("%d\t",brr1[i][j]);
        }
    }
    printf("\n\n");
}
```

OUTPUT

```
Transpose of a Matrix :
_____
Input the rows and columns of the matrix : 2
2
Input elements in the first matrix :
element - [0],[0] : 1
element - [0],[1] : 2
element - [1],[0] : 3
element - [1],[1] : 4

The matrix is :
1      2
3      4

The transpose of a matrix is :
1      3
2      4
```

Swap without temp variable

```
#include<stdio.h>
#include <conio.h>
void main ( )
{
    int a,b;
    clrscr( );
    printf(" \nEnter the value of a:");
    scanf("%d",&a);
    printf(" \nEnter the value of b:");
    scanf("%d",&b);
    a=a+b;
    b=a-b;
    a=a-b;
    printf(" \nThe value of a is:%d",a);
    printf(" \nThe value of b is:%d",b);
    getch( );
}
```

Output:

Enter the value of a:5
Enter the value of b:6

The value of a is:6
The value of b is:5

Swap with temp variable

```
#include<stdio.h>
#include <conio.h>
void main ( )
{
    int a,b,temp;
    clrscr( );
    printf(" \nEnter the value of a:");
    scanf("%d",&a);
    printf(" \nEnter the value of b:");
    scanf("%d",&b);
    temp=a;
    a=b;
    b=temp;
    printf(" \nThe value of a is:%d",a);
    printf(" \nThe value of b is:%d",b);
    getch( );
}
```

Output:

Enter the value of a:5
Enter the value of b:4

The value of a is:4
The value of b is:5

5) Sorting

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order.

Ranking of students is the process of sorting in descending order.

EMCET Ranking is an example for sorting with user-defined order.

EMCET Ranking is done with the following priorities.

i) First priority is marks obtained in EMCET.

ii) If marks are same, the ranking will be done with comparing marks obtained in the Mathematics subject.

iii) If marks in Mathematics subject are also same, then the date of births will be compared.

Internal Sorting :

If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting.

External Sorting :

It is applied to Huge amount of data that cannot be accommodated in memory all at a time. So data in disk or file is loaded into memory part by part. Each part that is loaded is sorted separately, and stored in an intermediate file and all parts are merged into one single sorted list.

Types of Internal Sorting's

Bubble Sort

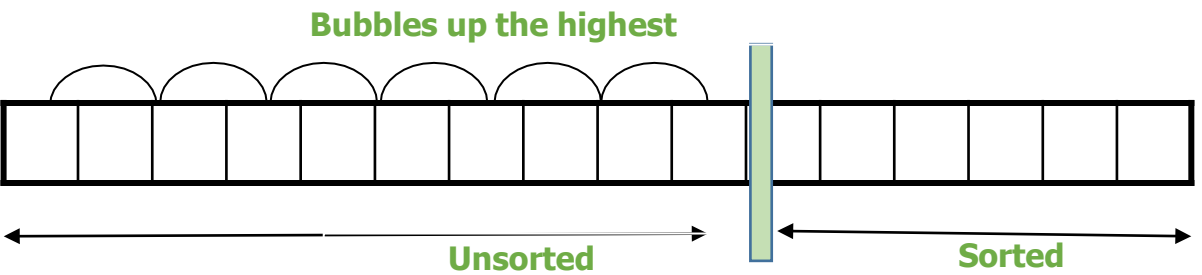
Insertion Sort

Selection Sort

Quick Sort

Merge Sort

Exercise Program 2: Sort the numbers using bubble sort



10	54	54	54	54	54
47	10	47	47	47	47
12	47	10	23	23	23
54	12	23	10	19	19
19	23	12	19	10	12
23	19	19	12	12	10
Original List	After Pass 1	After Pass 2	After Pass 3	After Pass 4	After Pass 5

```
Bubble_Sort ( A [ ], N )
Step 1 : Repeat For P = 1 to N - 1
Begin
Step 2 :   Repeat For J = 1 to N - P
Begin
Step 3 :     If ( A [ J ] < A [ J - 1 ] )
Swap ( A [ J ], A [ J - 1 ] )
)
End For
End For
Step 4 : Exit
```

Complexity of Bubble Sort

The complexity of sorting algorithm is depends upon the number of comparisons that are made.

Total comparisons in Bubble sort is

$$n (n - 1) / 2 \approx n^2 - n$$

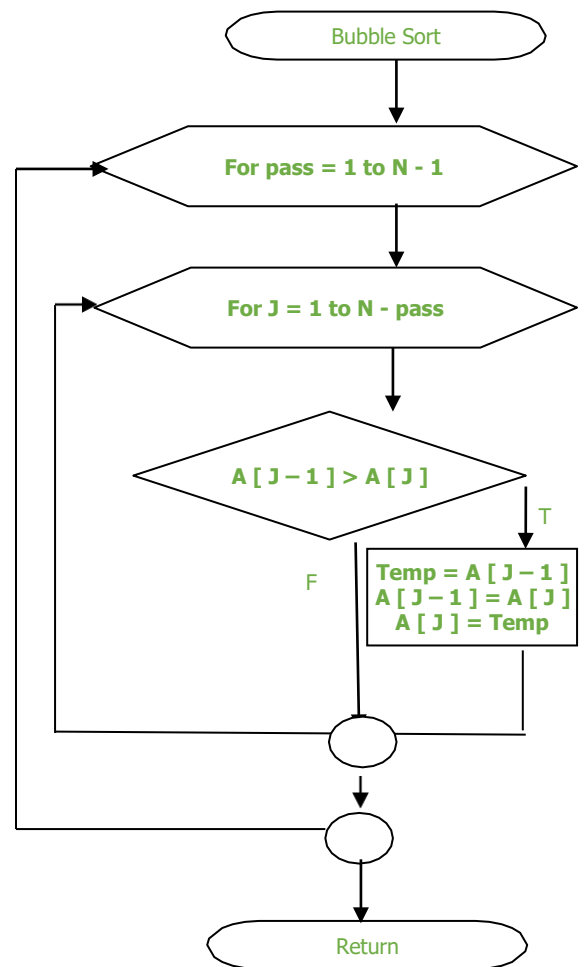
Complexity = $O (n^2)$

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

```
int main() {
    int count,num[50],i ;
    printf ("How many elements to be sorted
: ");
    scanf ("%d", &count);
    printf("Enter the elements : \n");
    for ( i = 0; i < count; i++) {
        printf ("num[%d] : ", i ); scanf( "%d",
        &num[ i ] );
    }
    printf("\n Array Before Sorting : \n");
    for (i=0;i<count ; i++)
        printf("%5d",num[i]);

    int pass, current, temp;
    for (pass=1;(pass<count);pass++) {
        for (current=1;current<=count-
        pass;current++){
            if (num[current-1]>num[current] )
            {
                temp = num[current-1];
                num[current-1] = num[current];
                num[current] = temp;
            }
        }
    }

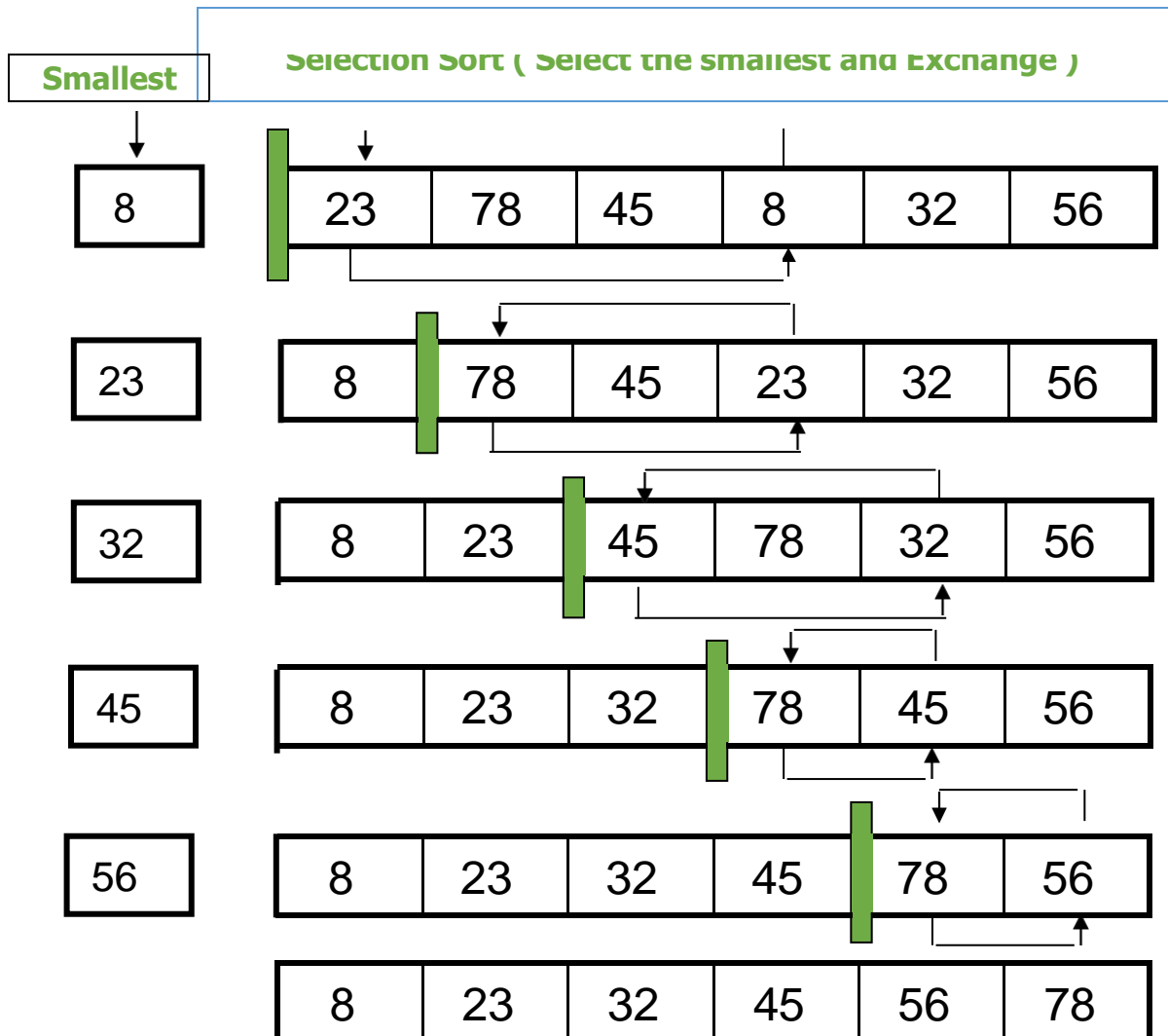
    printf("\nArray After Sorting : \n");
    for (i=0;i<count ; i++)
        printf("%5d",num[i]);
}
```



```
How many elements to be sorted : 5
Enter the elements :
num[0] : 5
num[1] : 9
num[2] : 1
num[3] : 3
num[4] : 7

Array Before Sorting :
    5    9    1    3    7
Array After Sorting :
    1    3    5    7    9
```

GUIDED ACTIVITY 2 – SELECTION SORT



```

Selection_Sort ( A [ ], N )
Step 1 : Repeat For K = 0 to N - 2
    Begin
Step 2 :   Set POS = K
Step 3 :   Repeat for J = K + 1 to N - 1
        Begin
            If A [ J ] < A [ POS ]
                Set POS = J
        End For
Step 5 :   Swap A [ K ] with A [ POS ]
    End For
Step 6 : Exit
    
```

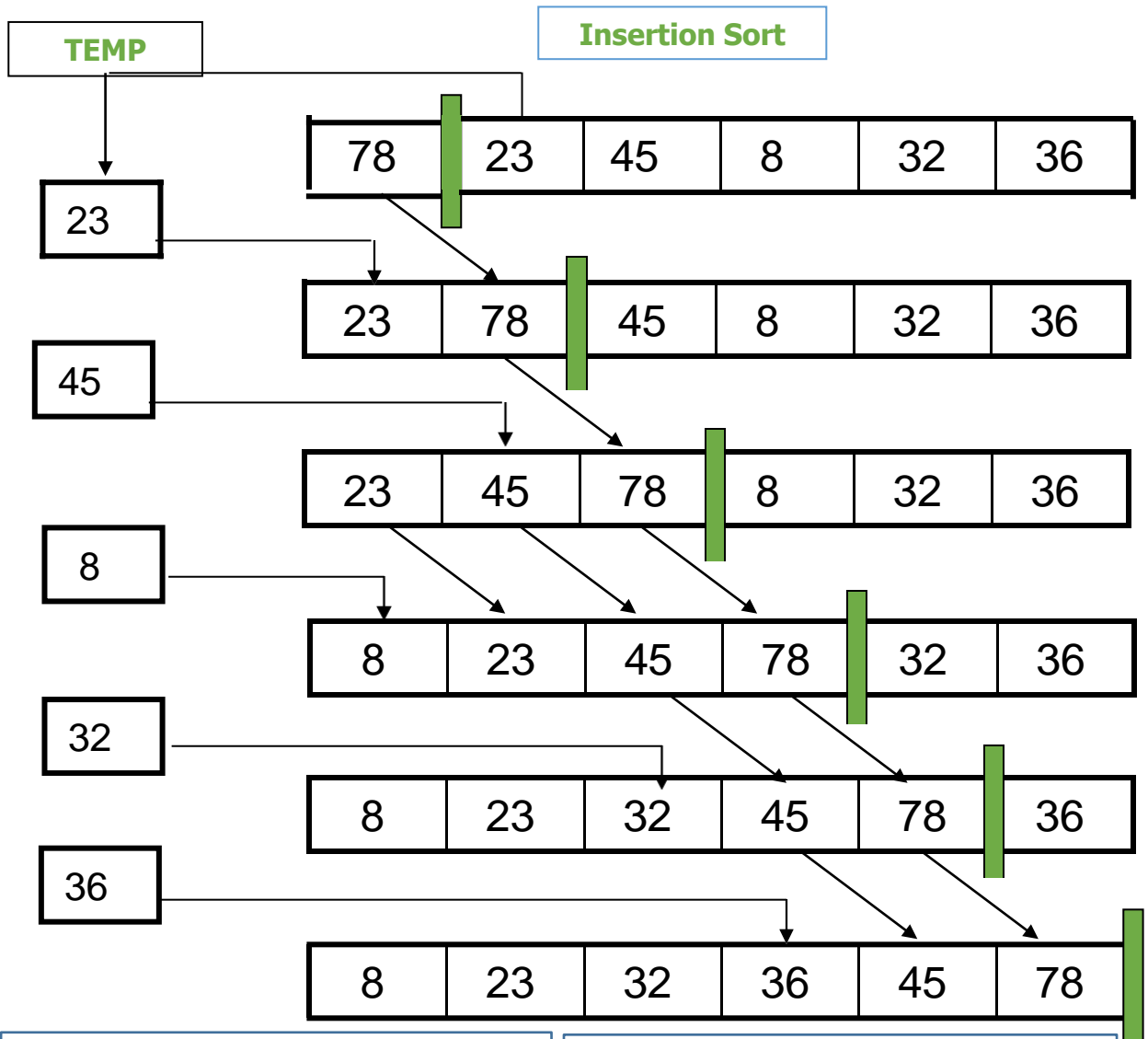
```

selection_sort ( int A [ ], int n ) {
    int k, j, pos, temp;
    for ( k = 0; k < n - 1; k++ ) {
        pos = k;
        for ( j = k + 1; j <= n; j++ ) {
            if ( A [ j ] < A [ pos ] )
                pos = j; }
        temp = A [ k ];
        A [ k ] = A [ pos ];
        A [ pos ] = temp;
    }
}
    
```

Complexity of Selection Sort

Best Case : $O (n^2)$
Average Case : $O (n^2)$
Worst Case : $O (n^2)$

GUIDED ACTIVITY 3 –INSERTION SORT



```

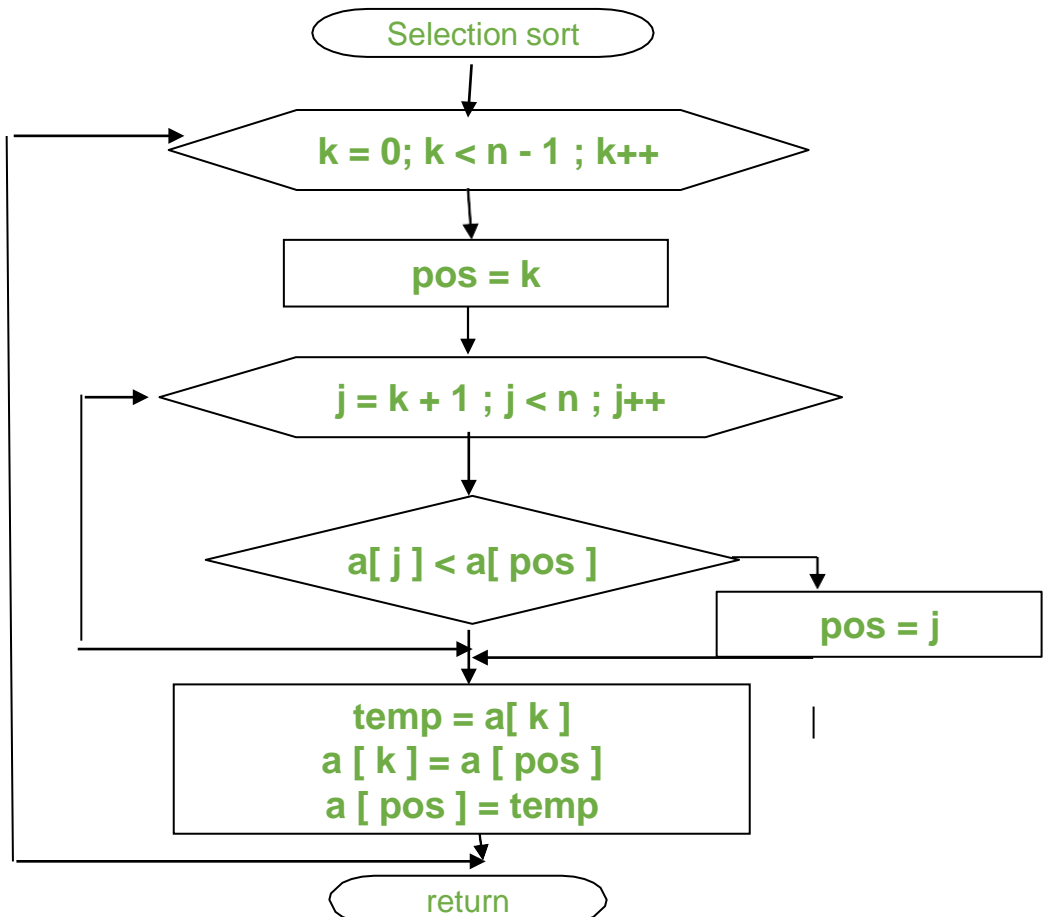
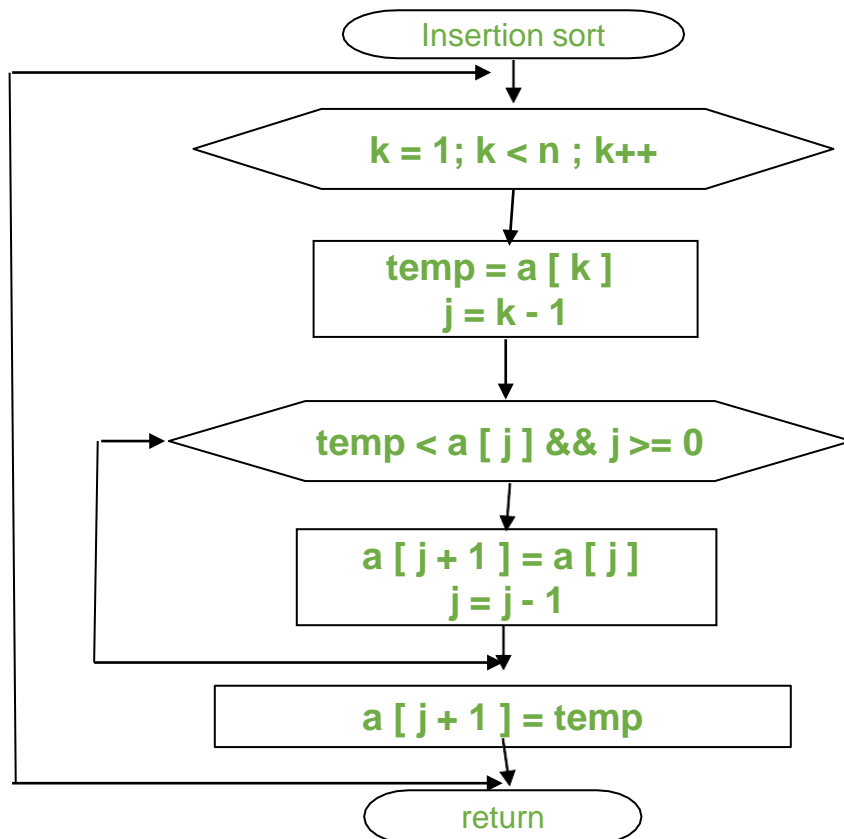
Insertion_Sort ( A [ ], N )
Step 1 : Repeat For K = 1 to N - 1
    Begin
Step 2 :   Set Temp = A [ K ]
Step 3 :   Set J = K - 1
Step 4 :   Repeat while Temp < A [ J ] AND J >= 0
        Begin
            Set A [ J + 1 ] = A [ J ]
            Set J = J - 1
        End While
Step 5 :   Set A [ J + 1 ] = Temp
    End For
Step 4 : Exit
    
```

```

insertion_sort ( int A [ ], int n ) {
    int k, j, temp;
    for ( k = 1; k < n; k++ ) {
        temp = A [ k ];
        j = k - 1;
        while ( ( temp < A [ j ] ) && ( j >= 0 ) ) {
            A [ j + 1 ] = A [ j ];
            j--;
        }
        A [ j + 1 ] = temp;
    }
}
    
```

Complexity of Insertion Sort

Best Case : $O(n)$
Average Case : $O(n^2)$
Worst Case : $O(n^2)$



Comparison : Bubble sort – Insertion sort – Selection sort

Bubble Sort :

- very primitive algorithm like linear search, and least efficient .
- No of swapping are more compare with other sorting techniques.
- It is not capable of minimizing the travel through the array like insertion sort.

Insertion Sort :

- sorted by considering one item at a time.
- efficient to use on small sets of data.
- twice as fast as the bubble sort.
- 40% faster than the selection sort.
- no swapping is required.
- It is said to be online sorting because it continues the sorting a list as and when it receives
new elements.
- it does not change the relative order of elements with equal keys.
- reduces unnecessary travel through the array.
- requires low and constant amount of extra memory space.
- less efficient for larger lists.

Selection sort :

- No of swapping will be minimized. i.e., one swap on one pass.
- generally used for sorting files with large objects and small keys.
- It is 60% more efficient than bubble sort and 40% less efficient than insertion sort.
- It is preferred over bubble sort for jumbled array as it requires less items to be exchanged.
- uses internal sorting that requires more memory space.
- It cannot recognize sorted list and carryout the sorting from the beginning, when new elements
are added to the list.

Topics to be covered – matrix operations (* addition, subtraction, multiplication, inverse, determinant, sum of principal diagonal element)

addition and subtraction of two Matrices

```
/*Program to add two add matrices*/
#include<stdio.h> // include stdio.h
#define ROW 2
#define COL 3
int main(){
    int i, j, arr1[ROW][COL], arr2[ROW][COL];
    printf("Enter first matrix: \n");
    for(i = 0; i < ROW; i++){
        for(j = 0; j < COL; j++){
            scanf("%d", &arr1[i][j]);}}

    printf("\nEnter second matrix: \n");
    for(i = 0; i < ROW; i++){
        for(j = 0; j < COL; j++){
            scanf("%d", &arr2[i][j]);}}
    printf("\narr1 + arr2 = \n");
    // add two matrices
    for(i = 0; i < ROW; i++){
        for(j = 0; j < COL; j++){
            printf("%5d ", arr1[i][j]+arr2[i][j]);}
        printf("\n");}
    return 0;}
```

How it Works

To add or subtract matrices we simply add or subtract corresponding entries in each matrix resp.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} + \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11} + B_{11} & A_{12} + B_{12} \\ A_{21} + B_{21} & A_{22} + B_{22} \end{pmatrix}$$

OUTPUT

```
Enter first matrix:
1 2 3
4 5 6

Enter second matrix:
2 4 6
8 10 12

arr1 + arr2 =
    3    6    9
   12   15   18
```

Multiplication of two matrices

```
/*Program to multiply two matrices*/
#include<stdio.h> // include stdio.h
#define ROW1 2
#define COL1 2
#define ROW2 COL1
#define COL2 3
int main(){
    int i, j, arr1[ROW1][COL1],
        arr2[ROW2][COL2],
        arr3[ROW1][COL2];
    printf("Enter first matrix (%d x %d): \n", ROW1, COL1);
    // input first matrix
    for(i = 0; i < ROW1; i++) {
        for(j = 0; j < COL1; j++) {
            scanf("%d", &arr1[i][j]); } }
    printf("\nEnter second matrix (%d x %d): \n", ROW2, COL2);
    // input second matrix
    for(i = 0; i < ROW2; i++){
        for(j = 0; j < COL2; j++){
            scanf("%d", &arr2[i][j]);} }
    printf("\narr1 * arr2 = ");
    // multiply two matrices
    for(i = 0; i < ROW1; i++) {
        for(j = 0; j < COL2; j++){
            arr3[i][j] = 0;
            for(int k = 0; k < COL1; k++){
                arr3[i][j] += arr1[i][k] * arr2[k][j]; } }
        printf("\n"); }
    // print the result
    for(i = 0; i < ROW2; i++) {
        for(j = 0; j < COL2; j++) {
            printf("%d ", arr3[i][j]); }
        printf("\n"); }
    return 0; }
```

How it Works

Two matrices can be multiplied only if the number of columns in the first matrix is equal to the number of rows in the second matrix.

Let A be the matrix of size 2x3 and B be the matrix of size 3x2. then, A*B is given by. In general, if matrix A is of size m×n, and B is of size n×p, then the size of matrix A*B will be m × p.

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} * \begin{pmatrix} g & h \\ k & l \\ o & p \end{pmatrix} = \begin{pmatrix} a*g+b*k+c*o & a*h+b*l+c*p \\ d*g+e*k+f*o & d*h+e*l+f*p \end{pmatrix}$$

```
Enter first matrix (2 x 2):
2 3
4 5
Enter second matrix (2 x 3):
6 4 2
7 8 9
arr1 * arr2 =
33 32 31
59 56 53
```


Determinant of a 2D matrix

For eg1. $a = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$ $|a| = ad - bc$

eg2. $a = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$ $|a| = a(ei - fh) - b(di - gf) + c(dh - eg)$

```
#include<stdio.h>
void main(){
int a[3][3],i,j,det;
printf("enter 3x3 matrix:\n");
for (i=0;i<3;i++){
for (j=0;j<3;j++){
scanf("%d",&a[i][j]);}}
det=a[0][0]*(a[1][1]*a[2][2]-a[2][1]*a[1][2])
-a[0][1]*(a[1][0]*a[2][2]-a[2][0]*a[1][2])
+a[0][2]*(a[1][0]*a[2][1]-a[2][0]*a[1][1]);
printf("\ndeterminant is %d", det);
}
```

enter 3x3 matrix:

1 2 3 4 5 6 7 8 9

determinant is 0

sum of principal diagonal element of a square matrix.

```
#include<stdio.h>
void main(){
int a[10][10],n,sum=0,i,j;
printf("Enter order of the square matrix:");
scanf("%d",&n);
printf("\nEnter matrix elements:\n");
for (i=0; i<n; i++) {
for (j=0; j<n; j++) {
scanf("%d",&a[i][j]); } }

for (i=0; i<n; i++)
sum=sum+a[i][i];
printf("sum of principal diagonal element is %d", sum);
}
```

Enter matrix elements:

1

2

3

4

sum of principal diagonal element is 5

Do it yourself : Inverse - assignment 1 – 2D array

Exercise Program 3 : Find whether given is matrix is diagonal or not.

/* Matrix Diagonal - Program to check whether a given matrix is diagonal matrix */
/* A diagonal matrix is that square matrix whose diagonal elements from upper left to lower right are non-zero and all other elements are zero. For example,

2 0 0
0 4 0
0 0 6

*/

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

```
void main(){
```

```
    int x[10][10], nr, nc, r, c, flag ;
```

```
    printf("Enter the number of rows and columns: ") ;
```

```
    scanf("%d %d", &nr, &nc) ;
```

```
    if(nr==nc){ /* checking for square matrix */
```

```
        printf("Enter elements of the matrix:\n") ;
```

```
        for(r=0 ; r<nr ; r++)
```

```
            for(c=0 ; c<nc ; c++)
```

```
                scanf("%d", &x[r][c]) ;
```

```
        flag=1 ;
```

```
        for(r=0 ; r<nr ; r++)
```

```
            for(c=0 ; c<nc ; c++)
```

```
                if(r==c){/*true for diagonal elements */
```

```
                    if(x[r][c]==0)
```

```
                        flag=0; }
```

```
                else{
```

```
                    if(x[r][c]!=0)
```

```
                        flag=0;}
```

```
        if(flag==1)
```

```
            printf("The matrix is diagonal") ;
```

```
        else
```

```
            printf("The matrix is not diagonal") ;
```

```
    }
```

```
else
```

```
    printf("The matrix is not a square matrix") ;}
```

```
Enter the number of rows and columns: 2
2
Enter elements of the matrix:
1
0
0
1
The matrix is diagonal
```

Multi dimensional array

A multi dimensional array is an array of arrays. Like we have one index in a single dimensional array, two indices in a two dimensional array, in the same way we have n indices in a n-dimensional array or multi dimensional array.

Conversely, an n dimensional array is specified using n indices.

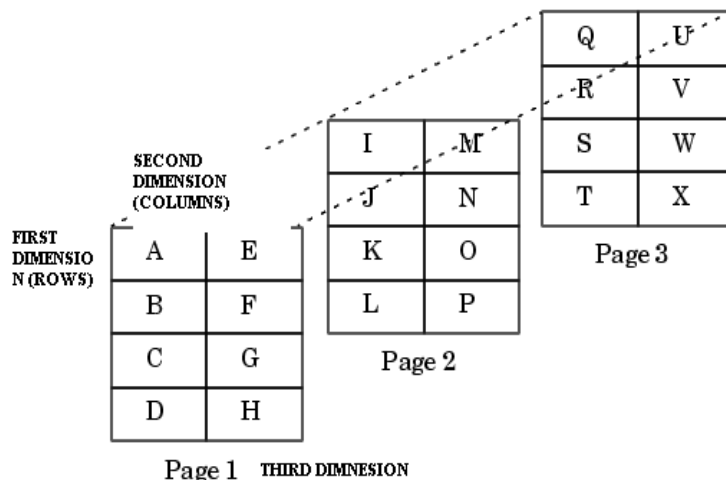
An n dimensional $m_1 \times m_2 \times m_3 \times \dots \times m_n$ array is a collection $m_1 \times m_2 \times m_3 \times \dots \times m_n$ elements. In a multi dimensional array, a particular element is specified by using n subscripts as **$A[I_1][I_2][I_3] \dots [I_n]$** , where,

$$I_1 \leq M_1$$

$$I_2 \leq M_2$$

$$I_3 \leq M_3$$

$$\dots \dots \dots I_n \leq M_n$$



Program To Read And Display A 2x2x2 Array

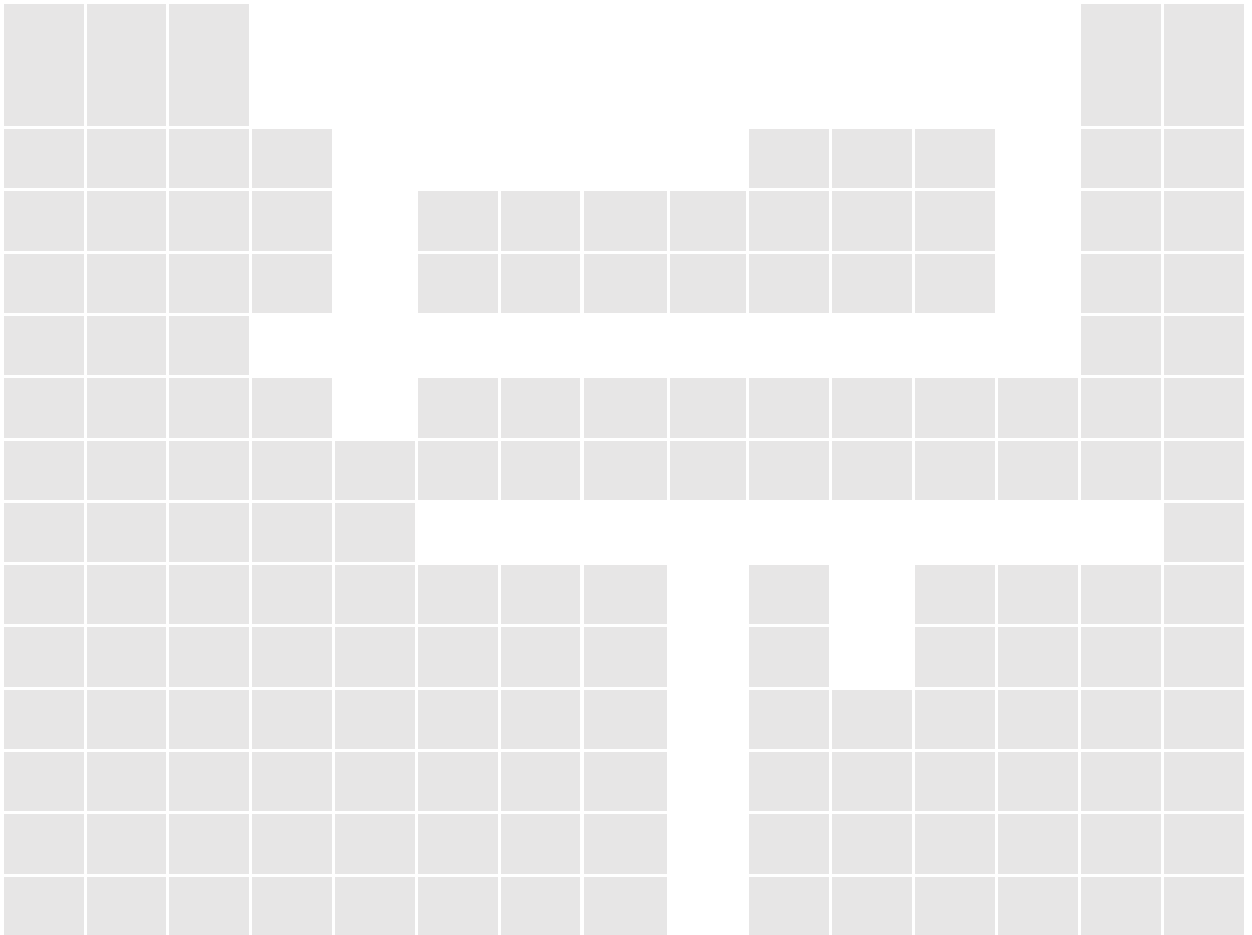
```
#include<stdio.h>
int main() { int array1[3][3][3], i, j, k;
    printf("\n Enter the elements of the matrix");
    printf("\n *****");
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            for(k=0;k<2;k++){
                printf("\n array[%d][ %d][ %d] = ", i, j, k);
                scanf("%d", &array1[i][j][k]);}
            }}
    printf("\n The matrix is : ");
    printf("\n *****")
    for(i=0;i<2;i++){
        printf("\n\n");
        for(j=0;j<2;j++){
            printf("\n");
            for(k=0;k<2;k++)
                printf("\t array[%d][ %d][ %d] = %d", i,
j, k, array1[i][j][k]);
            }}
}
```

OUTPUT

```
Enter the elements of the matrix
*****
array[0][ 0][ 0] = 1
array[0][ 0][ 1] = 2
array[0][ 1][ 0] = 1
array[0][ 1][ 1] = 2
array[1][ 0][ 0] = 1
array[1][ 0][ 1] = 2
array[1][ 1][ 0] = 1
array[1][ 1][ 1] = 2

The matrix is :
*****
array[0][ 0][ 0] = 1    array[0][ 0][ 1] = 2
array[0][ 1][ 0] = 1    array[0][ 1][ 1] = 2
array[1][ 0][ 0] = 1    array[1][ 0][ 1] = 2
array[1][ 1][ 0] = 1    array[1][ 1][ 1] = 2
```

GUIDED ACTIVITY 2 – Here is the crossword for you on Arrays



Across

- 1) Array[size] & Array[_____]
refer to the same element
- 2) Elements can be accessed
randomly by giving the
respective _____
- 3) Array consists of a set of
physically _____ memory
locations
- 4) The position of each array
element is known as array
index or _____
- 4)

Down

- 1) In array, each block ____
____ the same data type.
- 2) Language does not ____
array bound checking
- 3) 2D array can be declared
without specifying the
____ size
____ are a group of similar
elements.



Test Yourself – 2.2 Topics (Arrays in C - Two dimensional arrays)

1. 2D array initialized with Employee Id and Salary

```
int aiEmployeeInfo[][2]=  
{1001, 25000, 1002, 20000, 1003, 15000};
```

Write a printf statement to display 2nd Employee ID and salary from the above array declaration

2. float aiEmployeeInfo[2][5];

How many bytes are allocated for the above array declaration?

3. Int aiEmployeeInfo[2][5];

Which of the following is the CORRECT statement to read value to array element at row 0 and column 1.

Scanf(":%d", aiEmployeeInfo[0])

Scanf(":%d", &aiEmployeeInfo[0])

Scanf(":%d", aiEmployeeInfo[0][1])

Scanf(":%d", &aiEmployeeInfo[0][1])

4. Which of the following is a two-dimensional array?

A. array anarray[20][20];

B. int anarray[20][20];

C. int array[20, 20];

D. char array[20];

5. Consider the following array declared in C. Which of the following operations is the most appropriate for looping through the elements of the array? Int matrix[3][2][1];

A. While loop

B. Nested for loop (3 levels)

C. Nested for loop (2 levels)

D. Do... while



REVIEW questions and answers – 2.2 Topic (Arrays in C - Two dimensional arrays)

i) State Whether the following statement are true or false:

- a) In c, by default, the first subscript and second subscript is zero

True

- b) When initializing a multidimensional array, not specifying all its dimensions is an error.

True

- c) In C, we can use a maximum of 4 dimensions of an array

False

ii) Identify the errors, if any in each of declaration, by assumption ROW and COLUMN are declared as symbolic constants.

- a) Float average[ROW],[COLUMN];

Incorrect

- b) Long int number[ROW]

Incorrect

- c) int sum[][]

Incorrect

iii) Fill in the blanks in the following statements

- a) An array that uses more than two subscripts is referred to as ____ array.

Multidimensional

- b) ____ is the process of arranging the elements of an array in order.

Sorting

iv) Discussion on writing a for loop statement that initializes all the diagonal elements of an array to one and other to zero to be shown. Assume 5 rows and 5 columns.

```
for (i=0;i<5;i++)
    for (j=0;j<5;j++)
    {
        if (i==j)
            printf("1");
        else
            printf("0");
    }
```

Assignment

Unit II

Assignment Questions

CO 1	Develop C program solutions to simple computational problems		
1.	<p>Write a program in C to read n number of values in an array and display it in reverse order.</p> <p>Test Data :</p> <p>Input the number of elements to store in the array :3</p> <p>Input 3 number of elements in the array :</p> <p>element - 0 : 2</p> <p>element - 1 : 5</p> <p>element - 2 : 7</p> <p>Expected Output :</p> <p>The values store into the array are :</p> <p>2 5 7</p> <p>The values store into the array in reverse are :</p> <p>7 5 2</p>	K2	CO1
2.	<p>Write a program in C to merge two arrays of same size sorted in decending order. Test Data :</p> <p>Input the number of elements to be stored in the first array :3</p> <p>Input 3 elements in the array :</p> <p>element - 0 : 1</p> <p>element - 1 : 2</p> <p>element - 2 : 3</p> <p>Input the number of elements to be stored in the second array :3</p> <p>Input 3 elements in the array :</p> <p>element - 0 : 1</p> <p>element - 1 : 2</p> <p>element - 2 : 3</p> <p>Expected Output :</p> <p>The merged array in decending order is :</p> <p>3 3 2 2 1 1</p>	K2	CO1
3.	<p>Write a program in C to separate odd and even integers in separate arrays.</p> <p>Test Data :</p> <p>Input the number of elements to be stored in the array :5</p> <p>Input 5 elements in the array :</p> <p>element - 0 : 25</p> <p>element - 1 : 47</p> <p>element - 2 : 42</p> <p>element - 3 : 56</p> <p>element - 4 : 32</p> <p>Expected Output :</p> <p>The Even elements are :</p> <p>42 56 32</p> <p>The Odd elements are :</p> <p>25 47</p>	K2	CO1

Part A

Question & Answer

Part A

1) What is an Array in C language.? CO4)(K3)

A group of elements of same data type.

2) In genral what is correct statement about C language arrays.

An array address is the address of first element of array itself.

An array size must be declared if not initialized immediately.

Array size is the sum of sizes of all elements of the array.

3) What are the Types of Arrays.?

Types of arrays includes; A) int, long, float, double B) struct, enum and C) char

4) How do An array Index starts with?

It always starts with 0.

6) What is the output of C Program.? `int main() { int a[]; a[4] = {1,2,3,4}; printf("%d", a[0]); }`

Output will be a Compiler error

7) What is the output of C Program.? `int main() { int a[] = {1,2,3,4}; int b[4] = {5,6,7,8}; printf("%d,%d", a[0], b[0]); }`

Output will be 1,5

8) What is the output of C Program.? `int main() { char grade[] = {'A','B','C'}; printf("GRADE=%c, ", *grade); printf("GRADE=%d", grade); }`

Output will be GRADE=A, GRADE=some address of array

9) What is the output of C program.? `int main() { char grade[] = {'A','B','C'}; printf("GRADE=%d, ", *grade); printf("GRADE=%d", grade[0]); }`

The output will be 65 65

10) What is the output of C program.? `int main() { float marks[3] = {90.5, 92.5, 96.5}; int a=0; while(a<3) { printf("%.2f,", marks[a]); a++; } }`

The resultant value will be 90.5 92.5 96.5

11) What is the output of C Program.?

```
int main() { int a[3] = {10,12,14}; a[1]=20; int i=0; while(i<3) { printf("%d ", a[i]); i++; } }
```

The output will be 10 20 14

Explanation: a[i] is (i+1) element. So a[1] changes the second element.

12) What is an array Base Address in C language.?

Base address in c include A) Base address is the address of 0th index element.

B) An array b[] base address is &b[0]

C) An array b[] base address can be printed with printf("%d", b);

13) What is the output of C Program with arrays and pointers.?

```
void change(int[]); int main() { int a[3] = {20,30,40}; change(a); printf("%d %d", *a, a[0]); } void change(int a[]) { a[0] = 10; }
```

Output: 10 10

Explanation: Notice that function change() is able to change the value of a[0] of main(). It uses Call By Reference. So changes in called function affected the original values.

14) Define an 2D array in C with two difference egs.? (CO4)(K3)

C looks a two dimensional array as an array of a one dimensional array. The 2-D array be visualized as a rectangular grid of rows and columns.

15) What is multi-dimensional array? (CO4)(K3)

An array with more than one subscript is called multi-dimensional array. In General an array with n subscripts is called n-dimensional array.

Part B

Questions

Part B

1. What is an Array and How to create an Array, adv and disadv.of array. (CO4)(K3)
2. What will happen when you access the array more than its dimension? (CO4)(K3)
3. Define arrays. Explain the array types with an example program for each type. (CO4)(K3)
4. Why use arrays and need of an array with eg.? (CO4)(K3)
5. What are the limitations of arrays. (CO4)(K3)
6. Describe how to declare one dimensional array in detail. (CO4)(K3)
7. Can we change the size of an array at run time? (CO4)(K3)
8. Can you declare an array without assigning the size of an array? (CO4)(K3)
- 9.What is the default value of Array in detail and why so? (CO4)(K3)
- 10.How to print element of Array? (CO4)(K3)
12. What is a two dimensional array. Explain its declaration, assignment and various initialization methods with examples. (CO4)(K3)
13. Is it practically possible to implement multi-dimensional array. If so justify your answer. (CO4)(K3)
14. Write a C program to arrange the numbers in ascending order.. (CO4)(K3)
15. Write a C program to subtract two matrices and display the resultant matrices. (CO4)(K3)
16. Write a C program to search an element using binary search. (CO4)(K3)
17. Write a C program to sort the given names [bubble sort]. (CO4)(K3)
18. Write a program in C to count a total number of duplicate elements in an array. (CO4)(K3)

Test Data :

Input the number of elements to be stored in the array :3

Input 3 elements in the array :

element - 0 : 5

element - 1 : 1

element - 2 : 1

Expected Output :

Total number of duplicate elements found in the array is :

19. Advantages and disadvantages of Array? (CO4)(K3)
20. How to find the missing number in integer array of 1 to 100? (CO4)(K3)
21. How to find largest and smallest number in unsorted array? (CO4)(K3)
22. How to find all pairs on integer array whose sum is equal to given number? (CO4)(K3)
23. How to rearrange array in alternating positive and negative number? (CO4)(K3)
24. How to reverse array in place. (CO4)(K3)
25. Write a program to read and display the elements using 1-D array. (CO4)(K3)
26. Write a C program to sort the given array elements in Ascending order. Discuss with examples. (CO4)(K3)
27. Write a C program to find the largest and smallest element given in an array of elements. (CO4)(K3)
28. Write a C program to read N integers into an array A and to find the (i) sum of odd numbers, (ii) sum of even numbers, (iii) average of all numbers. Output the results computed with appropriate headings. (CO4)(K3)
29. Write a C program to search an element using linear and binary techniques. (CO4)(K3)
30. Write a C program for [consider integer data] (i) Bubble sort (ii) Linear search (CO4)(K3)
31. Write a C program to read N numbers into an array & perform Linear search (CO4)(K3)
32. Write an algorithm and develop a C program that reads N integer numbers and arrange them in ascending order using selection Sort (CO4)(K3)
33. Write a C program to print the sum of diagonal elements of 2-D matrix. (CO4)(K3)
34. Write an algorithm and develop a C program to search an integer from N numbers in ascending order using binary searching technique. (CO4)(K3)
35. Write a C program to multiply two matrices of different order. (CO4)(K3)
36. Write a C program to add 2 matrices of size n by n. (CO4)(K3)
37. How 2-D array elements are stored in memory/ Explain with example (CO4)(K3)
38. Perform scalar matrix multiplication. (CO4)(K3)
39. Check whether two matrices are equal or not. (CO4)(K3)
40. Sum of the main diagonal elements of a matrix. (CO4)(K3)
41. Find the sum of minor diagonal elements of a matrix.

42. Possible way to Identity matrix in C. (CO4)(K3)
43. Write a C program to Check the sparse matrix. (CO4)(K3)
44. Check the symmetric matrix. (CO4)(K3)
45. Find the sum of each row and column of a matrix.
(CO4)(K3)
46. Interchange diagonals of a matrix. (CO4)(K3)
47. Write a C program to determine the upper triangular matrix. (CO4)(K3)
48. Find a lower triangular matrix. (CO4)(K3)
49. Sum of the upper triangular matrix. (CO4)(K3)
50. Write a C program to Find the sum of a lower triangular matrix. (CO4)(K3)
51. Write a C program to find the transpose of a matrix.
(CO4)(K3)
52. Write a C program to Find determinant of a matrix.
(CO4)(K3)

Supportive Online Certification

Unit II

Certification Courses

⚙ NPTEL

Problem solving through Programming in C

<https://nptel.ac.in/courses/106/105/106105171/>

⚙ Coursera

1) C for Everyone: Structured Programming

<https://www.coursera.org/learn/c-structured-programming>

2) C for Everyone: Programming Fundamentals

<https://www.coursera.org/learn/c-for-everyone>

Real time Applications

Unit II

Arrays are used at many places in real life applications and some applications are listed here.

- ✿ 2D Arrays, generally called Matrices are mainly used in Image processing
- ✿ RGB image is a $n \times n \times 3$ array
- ✿ It is used in Speech Processing where each speech signal is an array of Signal Amplitudes
- ✿ **Stacks** are used for storing intermediate results in Embedded systems
- ✿ The filters that are used to remove noise in a recording are also arrays
- ✿ **Playfair-cipher** is an old encrypting algorithm that uses a 2D array of alphabets as key to encrypt/decrypt text.
- ✿ Every string that you see in this answer is an array of characters
- ✿ An array of strings that gives some meaning is a sentence.
- ✿ A simple question Paper is an array of numbered questions with each of them mapped to some marks/points
- ✿ If had written this answer with numbered bullets, the answer would consist an array of numbered bullet points. But now, the answer consists of an array of un-numbered bullet points.

Content beyond syllabus

Unit II

Content beyond syllabus

1) Sorting- Topics covered for Insertion and selection sort.

2) Analysis of algorithms can be done for searching and sorting

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis –

- ⚙ **Worst-case** – The maximum number of steps taken on any instance of size **a**.
- ⚙ **Best-case** – The minimum number of steps taken on any instance of size **a**.
- ⚙ **Average case** – An average number of steps taken on any instance of size **a**.
- ⚙ **Amortized** – A sequence of operations applied to the input of size **a** averaged over time.

Assessment Schedule

Unit II

Prescribed Text book & References

Unit II

Text books & References

TEXT BOOK

1. Reema Thareja, "Programming in C", Oxford University Press, Second Edition, 2016

REFERENCES:

1. Kernighan, B.W and Ritchie,D.M, "The C Programming language", Second Edition, Pearson Education, 2006
2. Paul Deitel and Harvey Deitel, "C How to Program", Seventh edition, Pearson Publication
3. Juneja, B. L and Anita Seth, "Programming in C", CENGAGE Learning India pvt. Ltd., 2011
4. Pradip Dey, Manas Ghosh, "Fundamentals of Computing and Programming in C", First Edition, Oxford University Press, 2009

Mini Project Suggestions

Unit II

- 1) Safe Standard Library Containers
- 2) Simple encryption algorithm
- 3) Univ. Students Record System

Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

OCS752

INTRODUCTION TO C

PROGRAMMING

Department: : Electrical and Electronics Engineering

Batch/Year: 2017-2021

Created by: Dr. S. Meenakshi and A.S. Vibith

Date: 21-08-2020

Table of Contents

- ✿ Course Objectives
- ✿ Syllabus
- ✿ Course Outcomes (Cos)
- ✿ CO-PO Mapping
- ✿ Lecture Plan
- ✿ Activity based learning
- ✿ Lecture notes
- ✿ Assignments
- ✿ Part A Q&A
- ✿ Part B Qs
- ✿ List of Supportive online Certification courses
- ✿ Real time applications in day to day life and to industry
- ✿ Contents beyond Syllabus
- ✿ Assessment Schedule (proposed and actual date)
- ✿ Prescribed Text Books & Reference Books
- ✿ Mini Project Suggestions

Course Objectives

OCS752 INTRODUCTION TO C PROGRAMMING

L T P

3 0 0 3

OBJECTIVES

- ✿ To develop C Programs using basic programming constructs
- ✿ To develop C programs using arrays and strings
- ✿ To develop applications in C using functions and structures

Syllabus

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C 3 0 0 3 **UNIT I**

INTRODUCTION

9

Structure of C program – Basics: Data Types – Constants – Variables – Keywords – Operators: Precedence and Associativity – Expressions – Input/output statements, Assignment statements – Decision-making statements – Switch statement – Looping statements – Pre-processor directives – Compilation process – Exercise Programs: Check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers

UNIT II ARRAYS

9

Introduction to Arrays – One dimensional arrays: Declaration – Initialization – Accessing elements – Operations: Traversal, Insertion, Deletion, Searching – Two dimensional arrays: Declaration – Initialization – Accessing elements – Operations: Read – Print – Sum – Transpose – Exercise Programs: Print the number of positive and negative values present in the array – Sort the numbers using bubble sort – Find whether the given is matrix is diagonal or not.

UNIT III STRINGS

9

Introduction to Strings – Reading and writing a string – String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse – Substring – Insertion – Indexing – Deletion – Replacement – Array of strings – Introduction to Pointers – Pointer operators – Pointer arithmetic – Exercise programs: To find the frequency of a character in a string – To find the number of vowels, consonants and white spaces in a given text – Sorting the names.

Unit IV FUNCTIONS

9

Introduction to Functions – Types: User-defined and built-in functions – Function prototype – Function definition – Function call – Parameter passing: Pass by value – Pass by reference – Built-in functions (string functions) – Recursive functions – Exercise programs: Calculate the total amount of power consumed by 'n' devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)

UNIT V STRUCTURES

9

Introduction to structures – Declaration – Initialization – Accessing the members – Nested Structures – Array of Structures – Structures and functions – Passing an entire structure – Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

TOTAL:45 PERIODS

Course Outcomes

- ✿ CO 1 - Develop algorithmic solutions to simple computational problems using basic constructs K1
- ✿ CO 2 - Develop simple applications in C using Control Constructs K2
- ✿ CO 3 - Design and implement applications using arrays K2
- ✿ CO 4 – Represent data using string and string operations K3
- ✿ CO 5 - Decompose a C program into functions and pointers K3
- ✿ CO 6 - Represent and write program using structure and union K3

CO – PO Mapping

CO	PO	Mapping Level	Justification
CO	PO	Mapping Level	Justification
CO1	PO1	2	Identify the data type and operators to solve the problem
CO1	PO2	2	Design the expression in an efficient way
CO1	PO3	2	Recognize the need of basic c Tokens-variables-constants
CO1	PO5	2	Apply the concept of control statements for simple solving the problem
CO1	PO12	1	Formulate the iterative statements for problem solving
CO2	PO1	3	Develop a complete program s for preprocessor directives
CO2	PO2	3	Recognize the implementation of simple problem solving with above concepts
CO3	PO3	2	Apply simple mathematical concepts for writing 1D arrays and its operations
CO3	PO5	2	Identify and formulate for the given problem using 2D and its operations
CO3	PO12	1	Design way of problem solving in Multi Dimensional arrays
CO4	PO1	3	Recognize the need of implementation in string
CO4	PO2	3	Apply logic to solve simple problem statement using string operations
CO4	PO3	3	Apply the knowledge to find the possible code for string manipulations
CO5	PO5	2	Identify the code for decomposition as function
CO5	PO12	1	Develop functions and reuse it whenever required to reduce the lines of code
CO5	PO1	2	Recognize the need of function concepts
CO5	PO2	2	Apply compound data knowledge to select any one
CO5	PO3	2	Apply the concept of pointers
CO5	PO5	2	Design and Develop program using the selected compound data
CO6	PO12	1	Recognize the need of structure
CO6	PO1	2	Apply the basic idea of handling with union
CO6	PO2	2	Identify the number of modes and operations on structure in detail
CO6	PO3	2	Develop programs using structu

Lecture Plan

Unit III

CO-PO/PSO MAPPING

COURSE OUTCOME	LEVEL OF COURSE OUTCOME	PROGRAM OUTCOME (PO)												PROGRAM SPECIFIC OUTCOME (PSO)			
		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	K1	2	2	2	-	2	-	-	-	-	-	-	1	1			
CO2	K2	3	3	2	-	2	-	-	-	-	-	-	1	1			
CO3	K2	3	3	3	-	2	-	-	-	-	-	-	1	1			
CO4	K3	2	2	2	-	2	-	-	-	-	-	-	1	1			
CO5	K3	2	2	2	-	-	-	-	-	-	-	-	1	1			
CO6	K3	3	3	3		2							1	1			

Unit III - STRINGS

S.No	Topics	No. of Periods	Proposed Date	Actual Lecture Date	Pertaining CO	Taxonomy Level	Mode of Delivery
1	Introduction to Strings - Reading and writing a string -	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
2	String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse –	1			CO1	K2	PPT, Chalk & Talk
3	Substring – Insertion – Indexing – Deletion – Replacement	1			CO1	K2	PPT, Chalk & Talk
4	Array of strings – Introduction to Pointers	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
5.6	Pointer operators – Pointer arithmetic -	2			CO1	K2	PPT, Chalk & Talk
7	Exercise programs: To find the frequency of a character in a string	1			CO1	K2	PPT, Chalk & Talk
8,9	To find the number of vowels, consonants and white spaces in a given text - Sorting the names.	2			CO1	K2	PPT, Chalk & Talk

Activity Based Learning

Unit III

Activity Based Learning

- ✿ Learn by solving problems – Tutorial Sessions can be conducted
 - Tutorial sessions available in Skillrack for practice
- ✿ Learn by questioning
- ✿ Learn by doing hands-on IN ONLINR / VIRTUAL LAB.

Lecture Notes

UNIT III STRINGS

9

Introduction to Strings - Reading and writing a string - String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse – Substring – Insertion – Indexing – Deletion – Replacement – Array of strings – Introduction to Pointers – Pointer operators – Pointer arithmetic - Exercise programs: To find the frequency of a character in a string - To find the number of vowels, consonants and white spaces in a given text - Sorting the names.

Introduction to Strings

- ✿ A string is a null-terminated character array. This means that after the last character, a null character ('\0') is stored to signify the end of the character array.
- ✿ For example:

`char c[] = "c string";` When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character `\0` at the end by default.

c		s	t	r	i	n	g	\0
---	--	---	---	---	---	---	---	----

How to declare a string?

- ✿ Here's how you can declare strings:
- ✿ `char str[size];`
- ✿ `char s[5];`

Here, we have declared a string of 5 characters.

s[0]	s[1]	s[2]	s[3]	s[4]

A string can be declared as a **character array** or with a **string pointer**.

How to initialize strings?

- ✿ You can initialize strings in a number of ways.
- ✿ `char c[] = "abcd";`
- ✿ `char c[5] = {'a','b','c','d','\0'};`
- ✿ **`char *c="abcd";`**

Let's take another example:

- ✿ `char c[5] = "abcde";`

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters. This is bad and you should never do this.

Assigning Values to Strings

- ✿ Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,
- ✿ `char c[100];`
- ✿ **`c = "C programming"; // Error! array type is not assignable.`**

Note: Use the `strcpy()` function to copy the string instead

<pre>char str[] = "HELLO";</pre>	<pre>char ch = 'H';</pre> <p>Here H is a character not a string. The character H requires only one memory location.</p>
<pre>char str[] = "H";</pre> <p>Here H is a string not a character. The string H requires two memory locations. One to store the character H and another to store the null character.</p>	<pre>char str[] = "";</pre> <p>Although C permits empty string, it does not allow an empty character.</p>

Difference between character storage and string storage

For example if we write,

```
char str[] = "HELLO";
```

We are declaring a character array with 5 characters namely, H, E, L, L and O.

Besides, a null character ('\0') is stored at the end of the string. So, the internal representation of the string becomes- HELLO'\0'. Note that to store a string of length 5, we need 5 + 1 locations (1 extra for the null character).

The name of the character array (or the string) is a pointer to the beginning of the string.

<code>str[0]</code>	1000	H
<code>str[1]</code>	1001	E
<code>str[2]</code>	1002	L
<code>str[3]</code>	1003	L
<code>str[4]</code>	1004	O
<code>str[5]</code>	1005	\0

Memory representation of a character array

We can also declare a string with size much larger than the number of elements that are initialized.

For example, consider the statement below.

```
char str [10] = "HELLO";
```

In such cases, the compiler creates an array of size 10; stores "HELLO" in it and finally terminates the string with a null character. Rest of the elements in the array are automatically initialized to NULL.

Now consider the following statements:

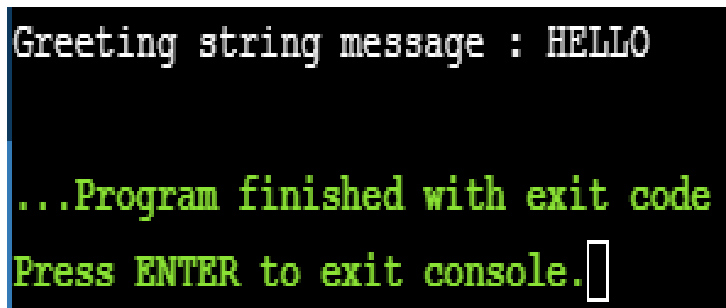
```
char str[3]; str = "HELLO";
```

The above initialization statement is illegal in C and would generate a compile-time error because of two reasons. First, the array is initialized with more elements than it can store. Second, initialization cannot be separated from declaration.

Note: When allocating memory space for a string, reserve space to hold the null character also.

Let us try to print above mentioned string:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char str[10]={'H','E','L','L','O','\0'};
    printf("Greeting string message : %s", str);
    return 0;
}
```



```
Greeting string message : HELLO

...Program finished with exit code
Press ENTER to exit console.█
```

When the above code is compiled and executed It produces result something as follows:

Note: %s is used to print the string in C

Reading and writing a string

READING STRINGS

If we declare a string by writing

```
char str[100];
```

Then str can be read from the user by using three ways

use scanf function

using gets() function

using getchar(), getch() or getche() function repeatedly

✿ The string can be read using scanf() by writing

```
scanf("%s", str);
```

Although the syntax of using scanf() function is well known and easy to use, the main pitfall of using this function is that the function terminates as soon as it finds a blank space (white space, newline, tab, etc.). For example, if the user enters Hello World, then the str will contain only Hello. This is because the moment a blank space is encountered, the string is terminated by the scanf() function. You may also specify a field width to indicate the maximum number of characters that can be read. Remember that extra characters are left unconsumed in the input buffer.

Unlike int, float, and char values, %s format does not require the ampersand before the variable str.

Note: Using & operand with a string variable in the scanf statement generates an error.

✿ The string can be read by writing

```
gets(str);
```

gets() is a simple function that overcomes the drawbacks of the scanf() function. gets() takes the starting address of the string which will hold the input. The string inputted using gets() is automatically terminated with a null character.

Note: that in this method, you have to deliberately append the string with a null character. The other two functions automatically do this

❁ The string can also be read by calling the `getchar()` repeatedly to read a sequence of single characters (unless a terminating character is entered) and simultaneously storing it in a character array.

```
i=0;
```

```
getchar(ch);
```

```
while(ch != '*')
```

```
{    str[i] = ch;
```

```
    i++;
```

```
    getchar(ch);
```

```
}    str[i] = '\0';
```

WRITING STRINGS

The string can be displayed on screen using three ways

- ✿ **use printf() function**
- ✿ **using puts() function**
- ✿ **using putchar()function repeatedly**

The string can be displayed using printf() by writing

```
printf("%s", str);
```

We use the format specifier %s to output a string. Observe carefully that there is no '&' character used with the string variable. We may also use width and precision specifications along with %s. The width specifies the minimum output field width. If the string is short, the extra space is either left padded or right padded. A negative width left pads short string rather than the default right justification. The precision specifies the maximum number of characters to be displayed, after which the string is truncated. For example,

```
printf ("%5.3s", str);
```

The above statement would print only the first three characters in a total field of five characters. Also these characters would be right justified in the allocated width. To make the string left justified, we must use a minus sign. For example,

```
printf ("%–5.3s", str);
```

The string can be displayed by writing

```
puts(str);
```

puts() is a simple function that overcomes the drawbacks of the printf() function.

Note: When the field width is less than the length of the string, the entire string will be printed, if the number of characters to be printed is specified as zero, then nothing is printed on the screen.

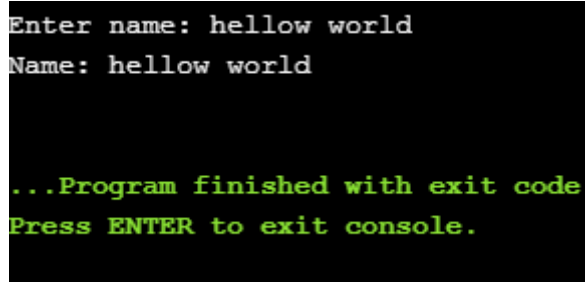
The string can also be written by calling the `putchar()` repeatedly to print a sequence of single characters

```
i=0;
while(str[i] !='\0')
{
    putchar(str[i]);
    i++;
}
```

Reading A Line Of Text

gets() and puts() are two string functions to take string input from user and display string respectively

```
int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name); //Function to read string from user.
    printf("Name: ");
    puts(name); //Function to display string.
    return 0;
}
```



```
Enter name: hellow world
Name: hellow world

...Program finished with exit code
Press ENTER to exit console.
```

Note: Though, gets() and puts() function handle strings, both these functions are defined in "stdio.h" header file.

SUPPRESSING INPUT

- ✿ `scanf()` can be used to read a field without assigning it to any variable. This is done by preceding that field's format code with a `*`. For example, given:

```
scanf("%d*c%d", &hr, &min);
```
- ✿ The time can be read as 9:05 as a pair. Here the colon would be read but not assigned to anything.

Using a Scanset

- ✿ The ANSI standard added the new **scanset** feature to the C language. A scanset is used to define a set of characters which may be read and assigned to the corresponding string. A scanset is defined by placing the characters inside square brackets prefixed with a `%`

- ✿ **int main()**

```
{  
    char str[10];  
    printf("\n Enter string: " );  
    scanf(""%[aeiou]", str );  
    printf( "The string is : %s", str);  
    return 0;  
}
```

- ✿ The code will stop accepting character as soon as the user will enter a character that is not a vowel.
- ✿ However, if the first character in the set is a `^` (caret symbol), then `scanf()` will accept any character that is not defined by the scanset. For example, if you write

```
scanf("%[^aeiou]", str );
```

String operations (using built-in string functions)

In this section, we will learn about different operations that can be performed **on strings using built in functions**.

C provides **string manipulating functions** in the "string.h" library.

String in C – Library Functions

Function	Purpose	Example
strlen	Returns the number of characters in a string	strlen("Hi") returns 2.
strlwr	Converts string to all lowercase	strlwr("Hi") returns hi.
strupr	Converts s to all uppercase	strupr("Hi");
strrev	Reverses all characters in s1 (except for the terminating null)	strrev(s1, "more");
strtok	Breaks a string into tokens by delimiters.	strtok("Hi, Chao", " ,");
strcpy	Makes a copy of a string	strcpy(s1, "Hi");
Strncpy	Copy the specified number of characters	strncpy(s1, "SVN",2);
strcat	Appends a string to the end of another string	strcat(s1, "more");
Strncat	Appends a string to the end of another string up to n characters	strncat(s1, "more",2);

Function	Purpose	Example
strcmp	Compare two strings alphabetically	strcmp(s1, "Hu");
Strncmp	Compare two string upto given n character	strncmp("mo", "more",2);
Stricmp	Compare two strings alphabetically without case sensitivity.	stricmp("hu", "Hu");
strchr()	Find first occurrence of a given character in the string	strchr(str1,c); Where c is the character variable
strrchr()	Find the last occurrence of a given character in the string	strrchr(str1,c)
strstr()	Finds the first occurrence of a given string in another string	strstr(str1,str2); Where str2 is the string to be searched in str1
strset()	sets all characters of a string to a given character	strset(str1,c);
strnset()	Sets first character of a string to a given character	Strnset(str1,c,n)

i) Length – reverse – upper case & lower case

Length of the string

The number of characters in the string constitutes the length of the string.

For example, `LENGTH("C PROGRAMMING IS FUN")` will return 20. Note that even blank spaces are counted as characters in the string. `LENGTH('0') = 0` and `LENGTH('') = 0` because both the strings does not contain any character.

`strlen()` function in C gives the length of the given string. Syntax for `strlen()` function is given below.

```
size_t strlen ( const char * str );
```

`strlen()` function counts the number of characters in a given string and returns the integer value.

It stops counting the character when null character is found. Because, null character indicates the end of the string in C.

Reversing a String

If `S1 = "HELLO"`, then reverse of `S1 = "OLLEH"`. To reverse a string, we just need to swap the first character with the last, second character with the second last character, and so on. Figure 4.7 shows an algorithm that reverses a string.

`strrev()` function reverses a given string in C language. Syntax for `strrev()` function is given below.

```
char *strrev(char *string);
```

`strrev()` function is non-standard function which may not available in standard library in C.

Upper Case of a String

`strupr()` function converts a given string into uppercase. Syntax for `strupr()` function is given below.

```
char *strupr(char *string);
```

`strupr()` function is non-standard function which may not available in standard library in C.

Lower Case of a String

`strlwr()` function converts a given string into lowercase. Syntax for `strlwr()` function is given below.

```
char *strlwr(char *string);
```

`strlwr()` function is non-standard function which may not available in standard library in C.

Example: C program to illustrate
strlen() , **strupr()** , **strlwr()** , **strrev()**

```
#include<stdio.h>
#include<string.h> //c header file for string library functions
void main(){
char str1[10]="DeNnis" , str2[10]="RitChiE";
int len;
len=strlen(str1);
//strupr(str1);
//strlwr(str2);
printf("\n Length of string is %d", len);
//printf("\n upper case is %s" , str1);
//printf("\n lower case is %s" ,str2);
strrev(str1);
printf("\n Reverse of string is %s", str1);
}
```

OUTPUT:

Length of string is 6

Reverse of string is sinNeD

Note: **strupr()** **strlwr()** - This is a non-standard function that works only with older versions of Microsoft C.

ii) Concatenate – copy & append

Concatenating two strings to form a new string

- ✿ `strcat()` function in C language concatenates two given strings. It concatenates source string at the end of destination string. Syntax for `strcat()` function is given below.
`char * strcat (char * destination, const char * source);`
- ✿ Example:
`strcat (str2, str1);` – `str1` is concatenated at the end of `str2`.
`strcat (str1, str2);` – `str2` is concatenated at the end of `str1`.
- ✿ As you know, each string in C is ended up with null character (`'\0'`).
- ✿ In `strcat()` operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after `strcat()` operation.

Copy One string to Another String

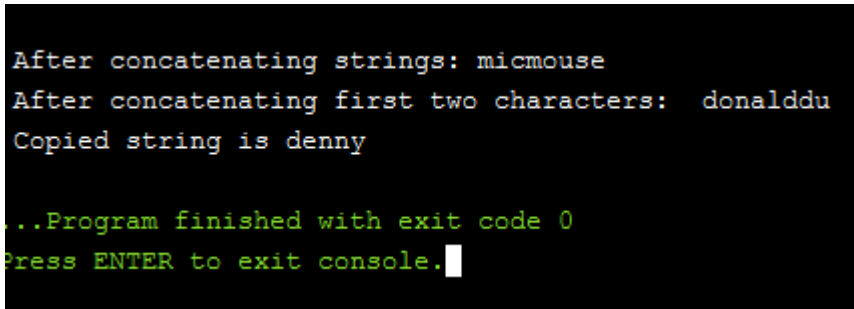
- ✿ `strcpy()` function copies contents of one string into another string. Syntax for `strcpy` function is given below.
`char * strcpy (char * destination, const char * source);`
- ✿ Example:
`strcpy (str1, str2)` – It copies contents of `str2` into `str1`.
`strcpy (str2, str1)` – It copies contents of `str1` into `str2`.
- ✿ If destination string length is less than source string, entire source string value won't be copied into destination string.
- ✿ For example, consider destination string length is 20 and source string length is 30. Then, only 20 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

Example: C program to illustrate

strcat() , strncat() , strcpy() , strncpy()

```
#include<stdio.h>
#include<string.h> //c header file for string library functions
void main(){
char str1[10]="mic" , str2[10]="mouse";
char str3[10]="donald" , str4[10]="duck";
char str5[10]="denny" , str6[10];
strcat(str1 ,str2);
printf("\n After concatenating strings: %s" , str1);
strncat(str3 ,str4 ,2); //appends first two char of str4 to str3
printf("\n After concatenating first two characters: %s" , str3);
strcpy(str6 ,str5);
printf("\n Copied string is %s" , str6);
}
```

OUTPUT:



```
After concatenating strings: micmouse
After concatenating first two characters: donalddu
Copied string is denney

...Program finished with exit code 0
Press ENTER to exit console.█
```

iii) Comparing two strings

Comparing the two strings

- ✿ If S1 and S2 are two strings, then comparing the two strings will give either of the following results:
 - ✿ S1 and S2 are equal
 - ✿ $S1 > S2$, when in dictionary order, S1 will come after S2
 - ✿ $S1 < S2$, when in dictionary order, S1 precedes S2
- ✿ To compare the two strings, each and every character is compared from both the strings. If all the characters are the same, then the two strings are said to be equal.
- ✿ strcmp() function in C compares two given strings and returns zero if they are same.
- ✿ If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value. Syntax for strcmp() function is given below.
`int strcmp (const char * str1, const char * str2);`
- ✿ strcmp() function is case sensitive. i.e, "A" and "a" are treated as different characters.

Example: C program to illustrate

strcmp() , stricmp() , strncmp() , strnicmp()

```
#include<stdio.h>
#include<string.h> //c header file for string library functions
void main(){
char str1[10]="Charles" , str2[10]="charles";
char str3[10]="charlie";
If (strcmp(str1 , str2)==0)
    printf("\n Equal strings");
else
    printf("\n Strings are different");
If (stricmp(str1 , str2)==0)
    printf("\n Equal strings");
else
    printf("\n Strings are different");
If (strncmp(str2 , str3 , 4)==0)
    printf("\n First four characters are same");
else
    printf("\n First four characters are different");
}
```

OUTPUT:

Strings are different

Equal strings

First four characters are different

iv) Substring /find out occurrence?

C substring program to find substring of a string and its all subrnrgs. A substring is itself a string that is part of a longer string. For example, substrings of string "the" are "" (empty string), "t", "th", "the", "h", "he" and "e."

`strchr()` , `strrchr()` , `strstr()`

`strchr()` function returns pointer to the first occurrence of the character in a given string. Syntax for `strchr()` function is given below.

```
char *strchr(const char *str, int character);
```

`strrchr()` function in C returns pointer to the last occurrence of the character in a given string. Syntax for `strrchr()` function is given below.

```
char *strrchr(const char *str, int character);
```

`strstr()` function returns pointer to the first occurrence of the string in a given string. Syntax for `strstr()` function is given below.

```
char *strstr(const char *str1, const char *str2);
```

Note:

The header file "string.h" does not contain any library function to find a substring directly like `substr()` but can find either using `strchr` /`strnchr`/`strstr()` or by manually by applying logic and similarly for Insertion and Deletion as well.

Example: C program to illustrate strchr() , strrchr() , strstr()

```
#include<stdio.h>
#include<string.h> //c header file for string library functions
void main(){
    char str1[15]="Miscky Vickys";
    printf("\n Using strchr : %s" , strchr(str1 , 'i'));
    printf("\n Using strrchr : %s" , strrchr(str1 , 'i'));
    printf("\n Using strstr  : %s" , strstr(str1 , 'ky'));
}
```

OUTPUT:

Using strchr : iscky Vickys

Using strrchr : iscky

Using strstr : ky Vickys

v) Indexing

vi) replacement

strset() function sets all the characters in a string to given character.

Syntax for strset() function is given below.

```
char *strset(char *string, int c);
```

strnset() function sets portion of characters in a string to given character. Syntax for strnset() function is given below.

```
char *strnset(char *string, int c);
```

strnset() function is non standard function which may not available in standard library in C.

C program to illustrate strset() , strnset()

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

```
#include<string.h> //c header file for string library functions
```

```
void main(){
```

```
    char str1[15]="mypassword";
```

```
    char str2[15]="8754538560";
```

```
    strnset(str2, '*');
```

```
    strnset(str2, '*', 8);
```

```
    printf("\n Using strset  : %s" , str1);
```

```
    printf("\n Using strnset : %s" , str2);
```

```
}
```

OUTPUT:

```
Using strset  : *****
```

```
Using strnset : *****60
```

Note: strset() function is non standard function which may not available in standard library in C.

String operations (without using string built-in/lib functions)

In this section, we will learn about different operations that can be performed on strings without using built in functions which includes :

i) Length – ii) Compare – *iii) Concatenate – *iv) Copy – v) Reverse – vi) Substring – vii) Insertion – viii) Indexing – ix) Deletion – x) Replacement

i) Length - Finding Length of a String

The number of characters in the string constitutes the length of the string.

For example, LENGTH("C PROGRAMMING IS FUN") will return 20.

LENGTH('') = 0 and LENGTH(' ') = 0 because both the strings does not contain any character.

```
Step 1: [INITIALIZE] SET I = 0
Step 2: Repeat Step 3 while STR[I] != NULL
Step 3:     SET I = I + 1
          [END OF LOOP]
Step 4: SET LENGTH = I
Step 5: END
```

Algorithm to calculate the length of a string

shows an algorithm that calculates the length of a string. In this algorithm, I is used as an index for traversing string STR. To traverse each and every character of STR, we increment the value of I. Once we encounter the null character, the control jumps out of the while loop and the length is initialized with the value of I.

Note: that even blank spaces are counted as characters in the string and for remembering in this string – index is from 0 and position is from 1

index = position-1 or index+1 = position

Write a program to find the length of a string.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char str[100], i = 0, length=0;
    printf("\n Enter the string : ");
    gets(str);
    /*while(str[i] != '\0') // using while loop
        i++;
    length = i;*/

    for (i=0;str[i]!='\0';i++)
        length = length+1;

    printf("\n The length of the string is : %d", length);
    return 0;
}
```

OUTPUT

```
Enter the string : hellow
The length of the string is : 6
```


ii) Compare

If S1 and S2 are two strings, then comparing the two strings will give either of the following results:

S1 and S2 are equal

S1>S2, when in dictionary order, S1 will come after S2

S1<S2, when in dictionary order, S1 precedes S2

To compare the two strings, each and every character is compared from both the strings. If all the characters are the same, then the two strings are said to be equal. Figure shows an algorithm that compares two strings.

In this algorithm, we first check whether the two strings are of the same length. If not, then there is no point in moving ahead, as it straight away means that the two strings are not the same. However, if the two strings are of the same length, then we compare character by character to check if all the characters are same. If yes, then the variable SAME is set to 1. Else, if SAME = 0, then we check which string precedes the other in the dictionary order and print the corresponding message.

```
Step 1: [INITIALIZE] SET I=0, SAME =0
Step 2: SET LEN1 = Length(STR1), LEN2 = Length(STR2)
Step 3: IF LEN1 != LEN2
        Write "Strings Are Not Equal"
    ELSE
        Repeat while I<LEN1
            IF STR1[I] == STR2[I]
                SET I = I + 1
            ELSE
                Go to Step 4
            [END OF IF]
        [END OF LOOP]
        IF I = LEN1
            SET SAME =1
            Write "Strings are Equal"
        [END OF IF]
Step 4: IF SAME = 0,
        IF STR1[I] > STR2[I]
            Write "String1 is greater than String2"
        ELSE IF STR1[I] < STR2[I]
            Write "String2 is greater than String1"
        [END OF IF]
    [END OF IF]
Step 5: EXIT
```

Algorithm to compare two strings

Write a program to compare two strings.

```
#include <stdio.h> #include <conio.h> #include <string.h>
int main()
{
char str1[50], str2[50];
int i=0, len1=0, len2=0, same=0; clrscr();
printf("\n Enter the first string : "); gets(str1);
printf("\n Enter the second string : "); gets(str2);
len1 = strlen(str1); len2 = strlen(str2); if(len1 == len2)
{
while(i<len1)
{
if(str1[i] == str2[i])
i++;
else break;
}
if(i==len1)
{
same=1;
printf("\n The two strings are equal");
}
}
if(len1!=len2)
printf("\n The two strings are not equal"); if(same == 0)
{
if(str1[i]>str2[i])
printf("\n String 1 is greater than string 2"); else if(str1[i]<str2[i])
printf("\n String 2 is greater than string 1");
}
getch(); return 0;
}
```

OUTPUT:

Enter the first string : Hello Enter the second string : Hello The two strings are equal

```
#include <stdio.h>
int main()
{
    int i, diff=0;
    char str1[100], str2[100];
    printf("\n Enter the string1 : ");
    gets(str1);
    printf("\n Enter the string2 : ");
    gets(str2);
    for (i=0;str1[i]!='\0' || str2[i]!='\0';i++)
        if (str1[i]!=str2[i])
        {
            diff=str1[i]-str2[i];
            break;
        }
    if (diff==0)
        printf("Strings are equal");
    else
        printf("Strings are not equal and their diff is %d",diff);
    return 0;
}
```

OUTPUT

```
Enter the string1 : Dennis
Enter the string2 : Ritchie
Strings are not equal and their diff is -14
```

iii) Concatenate

CONCATENATING TWO STRINGS TO FORM A NEW STRING

IF S1 and S2 are two strings, then concatenation operation produces a string which contains characters of S1 followed by the characters of S2.

```
ALGORITHM TO CONCATENATE TWO STRINGS

1. Initialize I =0 and J=0
2. Repeat step 3 to 4 while I <= LENGTH(str1)
3     SET new_str[J] = str1[I]
4     Set I =I+1 and J=J+1
   [END of step2]
5. SET I=0
6 Repeat step 6 to 7 while I <= LENGTH(str2)
7     SET new_str[J] = str1[I]
8     Set I =I+1 and J=J+1
   [END of step5]
9. SET new_str[J] = '\0'
10. EXIT
```

ALGORITHM TO CONCATENATE TWO STRINGS

```
#include <stdio.h>
int main()
{
    int i,j=0,len=0;
    char str1[100], str2[100];
    printf("\nEnter the string1 : ");
    gets(str1);
    printf("Enter the string2 : ");
    gets(str2);
    for (i=0;str1[i]!='\0';i++)
        len=len+1; // lengthof the first string
    j=len;
    for (i=0;str2[i]!='\0';i++){
        str1[j]=str2[i]; // 2nd string copied to 1st string //from jth
        position
        j=j+1;}
    str1[j]='\0';

    printf("The concatenated string is %s",str1);
}
```

OUTPUT

```
Enter the string1 : DEnnIS
Enter the string2 : Ritchie
The concatenated string is DEnnISRitchie
```

Note : concatenate or append have same implementa

Appending a String to Another String

Appending one string to another string involves copying the contents of the source string at the end of the destination string. For example, if S1 and S2 are two strings, then appending S1 to S2 means we have to add the contents of S1 to S2. So, S1 is the source string and S2 is the destination string. The appending operation would leave the source string S1 unchanged and the destination string $S2 = S2 + S1$.

Figure shows an algorithm that appends two strings.

In this algorithm, we first traverse through the destination string to reach its end, that is, reach the position where a null character is encountered. The characters of the source string are then copied into the destination string starting from that position. Finally, a null character is added to terminate the destination string.

```
Step 1: [INITIALIZE] SET I=0 and J=0
Step 2: Repeat Step 3 while DEST_STR[I] != NULL
Step 3:     SET I = I + 1
        [END OF LOOP]
Step 4: Repeat Steps 5 to 7 while SOURCE_STR[J] != NULL
Step 5:     DEST_STR[I] = SOURCE_STR[J]
Step 6:     SET I = I + 1
Step 7:     SET J = J + 1
        [END OF LOOP]
Step 8: SET DEST_STR[I] = NULL
Step 9: EXIT
```

Algorithm to append a string to another string

Write a program to append a string to another string.

```
#include <stdio.h>
#include <conio.h>
int main()
{
char Dest_Str[100], Source_Str[50];
int i=0, j=0;
printf("\n Enter the source string : ");
gets(Source_Str);
printf("\n Enter the destination string : ");
gets(Dest_Str);

while(Dest_Str[i] != '\0')
i++;
while(Source_Str[j] != '\0')
{
Dest_Str[i] = Source_Str[j];
i++;
j++;
}
Dest_Str[i] = '\0';
printf("\n After appending, the destination string is : ");
puts(Dest_Str);
return 0;
}
```

OUTPUT

Enter the source string : How are you?

Enter the destination string : Hello,

After appending, the destination string is : Hello,How are you?

iv) Copy

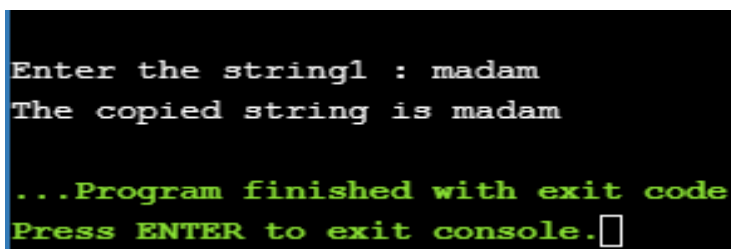
Copying the contents of one string to another string.

```
#include <stdio.h>
int main()
{
    int i;
    char str1[100], str2[100];
    printf("\n Enter the string1 : ");
    gets(str1);

    for (i=0;str1[i]!='\0';i++)
        str2[i]=str1[i];

    str2[i]='\0';
    printf("The copied string is %s",str2);
    return 0;
}
```

OUTPUT



```
Enter the string1 : madam
The copied string is madam

...Program finished with exit code
Press ENTER to exit console. □
```

v) Reverse

If $S1 = \text{"HELLO"}$, then reverse of $S1 = \text{"OLLEH"}$. To reverse a string, we just need to swap the first character with the last, second character with the second last character, and so on.

Figure shows an algorithm that reverses a string.

In Step 1, I is initialized to zero and J is initialized to the length of the string -1 . In Step 2, a while loop is executed until all the characters of the string are accessed. In Step 4, we swap the i th character of STR with its j th character. As a result, the first character of STR will be replaced with its last character, the second character will be replaced with the second last character of STR , and so on. In Step 4, the value of I is incremented and J is decremented to traverse STR in the forward and backward directions, respectively.

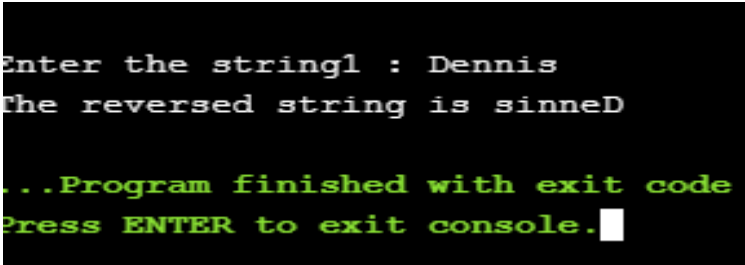
```
Step 1: [INITIALIZE] SET  $I=0$ ,  $J= \text{Length}(STR)-1$   
Step 2: Repeat Steps 3 and 4 while  $I < J$   
Step 3:   SWAP( $STR(I)$ ,  $STR(J)$ )  
Step 4:   SET  $I = I + 1$ ,  $J = J - 1$   
          [END OF LOOP]  
Step 5: EXIT
```

Algorithm to reverse a string

Write a program to reverse a given string.

```
#include <stdio.h>
int main()
{
    int i,j=0,len=0;
    char str1[100], rev[100];
    printf("\nEnter the string1 : ");
    gets(str1);
    for (i=0;str1[i]!='\0';i++)
        len=len+1;
    for (i=len-1;i>=0;i--)
    {
        rev[j]=str1[i];
        j=j+1;
    }
    rev[j]='\0';
    printf("The reversed string is %s",rev);
    return 0;
}
```

Outpt



```
Enter the string1 : Dennis
The reversed string is sinneD

...Program finished with exit code
Press ENTER to exit console. █
```

vi) Substring

To extract a substring from a given string, we need the following three parameters:

the main string,

the position of the first character of the substring in the given string,
and

the maximum number of characters/length of the substring.

For example, if we have a string

`str[] = "Welcome to the world of programming";`

Then, `SUBSTRING(str, 15, 5) = world`

Figure shows an algorithm that extracts a substring from the middle of a string.

In this algorithm, we initialize a loop counter I to M, that is, the position from which the characters have to be copied. Steps 3 to 6 are repeated until N characters have been copied. With every character copied, we decrement the value of N. The characters of the string are copied into another string called the SUBSTR. At the end, a null character is appended to SUBSTR to terminate the string.

```
Step 1: [INITIALIZE] Set I=M, J=0
Step 2: Repeat Steps 3 to 6
        while STR[I] != NULL and N>0
Step 3: SET SUBSTR[J] = STR[I]
Step 4: SET I = I + 1
Step 5: SET J = J + 1
Step 6: SET N = N - 1
        [END OF LOOP]
Step 7: SET SUBSTR[J] = NULL
Step 8: EXIT
```

Algorithm to extract a substring from
the middle of a string

Write a program to extract a substring from the middle of a given string.

```
#include <stdio.h>
#include <conio.h>
int main()
{
char str[100], substr[100];
int i=0, j=0, n, m;
printf("\n Enter the main string : ");
gets(str);
printf("\n Enter the position from which to start the substring: ");
scanf("%d", &m);
printf("\n Enter the length of the substring: ");
scanf("%d", &n);

i=m;
while(str[i] != '\0' && n>0)
{
substr[j] = str[i];
i++; j++; n--;
}
substr[j] = '\0';
printf("\n The substring is : "); puts(substr);
return 0;
}
```

OUTPUT

Enter the main string : Hi there

Enter the position from which to start the substring: 1

Enter the length of the substring: 4

The substring is : i th

viii) Indexing –program

Pattern Matching

This operation returns the position in the string where the string pattern first occurs. For example,

`INDEX("Welcome to the world of programming", "world") = 15`

However, if the pattern does not exist in the string, the INDEX function returns 0.

Figure shows an algorithm to find the index of the first occurrence of a string within a given text

In this algorithm, MAX is initialized to $\text{length}(\text{TEXT}) - \text{Length}(\text{STR}) + 1$.

For example, if a text contains 'Welcome To Programming' and the string contains 'World', in the main text, we will look for at the most $22 - 5 + 1 = 18$ characters because after that there is no scope left for the string to be present in the text.

Steps 3 to 6 are repeated until each and every character of the text has been checked for the occurrence of the string within it. In the inner loop in Step 3, we check the n characters of string with the n characters of text to find if the characters are same. If it is not the case, then we move to Step 6, where I is incremented. If the string is found, then the index is initialized with I, else it is set to -1.

For example, if `TEXT = WELCOME TO THE WORLD`

`STRING = COME`

In the first pass of the inner loop, we will compare COME with WELC character by character. As W and C do not match, the control will move to Step 6 and then ELCO will be compared with COME. In the fourth pass, COME will be compared with COME.

We will be using the programming code of pattern matching operation in the operations that follow.

```
Step 1: [INITIALIZE] SET I=0 and MAX = Length(TEXT)-Length(STR)+1
Step 2: Repeat Steps 3 to 6 while I < MAX
Step 3:   Repeat Step 4 for K = 0 To Length(STR)
Step 4:       IF STR[K] != TEXT[I + K], then Goto step 6
              [END OF INNER LOOP]
Step 5:   SET INDEX = I. Goto Step 8
Step 6:   SET I = I+1
              [END OF OUTER LOOP]
Step 7: SET INDEX = -1
Step 8: EXIT
```

Algorithm to find the index of the first occurrence of a string within a given text

vii) Insertion

The insertion operation inserts a string *S* in the main text *T* at the *k*th position. The general syntax of this operation is INSERT(text, position, string). For example, INSERT("XYZXYZ", 3, "AAA") = "XYZAAAXYZ"

Figure shows an algorithm to insert a string in a given text at the specified position.

This algorithm first initializes the indices into the string to zero. From Steps 3 to 5, the contents of NEW_STR are built. If *I* is exactly equal to the position at which the substring has to be inserted, then the inner loop copies the contents of the substring into NEW_STR. Otherwise, the contents of the text are copied into it.

```
Step 1: [INITIALIZE] SET I=0, J=0 and K=0
Step 2: Repeat Steps 3 to 4 while TEXT[I] != NULL
Step 3: IF I = pos
        Repeat while Str[K] != NULL
            new_str[J] = Str[K]
            SET J=J+1
            SET K = K+1
        [END OF INNER LOOP]
    ELSE
        new_str[J] = TEXT[I]
        set J = J+1
    [END OF IF]
Step 4: set I = I+1
        [END OF OUTER LOOP]
Step 5: SET new_str[J] = NULL
Step 6: EXIT
```

Algorithm to insert a string in a given text at the specified position

Write a program to insert a string in the main text.

```
#include <stdio.h>

int main() {
    char text[100], str[20], ins_text[100]; int i=0, j=0, k=0, pos;
    printf("\n Enter the main text : ");
    gets(text);
    printf("\n Enter the string to be inserted : ");
    gets(str);
    printf("\n Enter the place at which the string has to be inserted: ");
    scanf("%d", &pos);
    while(text[i] != '\0'){
        if(i==pos) {
            while(str[k] != '\0') {
                ins_text[j] = str[k];
                j++;
                k++;
            }
        } else {
            ins_text[j] = text[i];
            j++;
        }
        i++;
    }
    ins_text[j] = '\0';
    printf("\n The new string is : ");
    puts(ins_text);
    return 0;}
```

OUTPUT

Enter the main text : newsman

Enter the string to be inserted : paper

Enter the place at which the string has to be inserted: 4

The new string is: newspaperman

ix) Deletion –

Deleting a Substring from the Main String

The deletion operation deletes a substring from a given text. We can write it as DELETE(text, position, length). For example, DELETE("ABCDXXXABCD", 4, 3) = "ABCDABCD"

Figure shows an Algorithm to delete a substring from a text

In this algorithm, we first initialize the indices to zero. Steps 3 to 6 are repeated until all the characters of the text are scanned. If I is exactly equal to M (the position from which deletion has to be done), then the index of the text is incremented and N is decremented. N is the number of characters that have to be deleted starting from position M. However, if I is not equal to M, then the characters of the text are simply copied into the NEW_STR.

```
Step 1: [INITIALIZE] SET I=0 and J=0
Step 2: Repeat Steps 3 to 6 while TEXT[I] != NULL
Step 3: IF I=M
           Repeat while N>0
               SET I = I+1
               SET N = N - 1
           [END OF INNER LOOP]
       [END OF IF]
Step 4: SET NEW_STR[J] = TEXT[I]
Step 5: SET J = J + 1
Step 6: SET I = I + 1
       [END OF OUTER LOOP]
Step 7: SET NEW_STR[J] = NULL
Step 8: EXIT
```

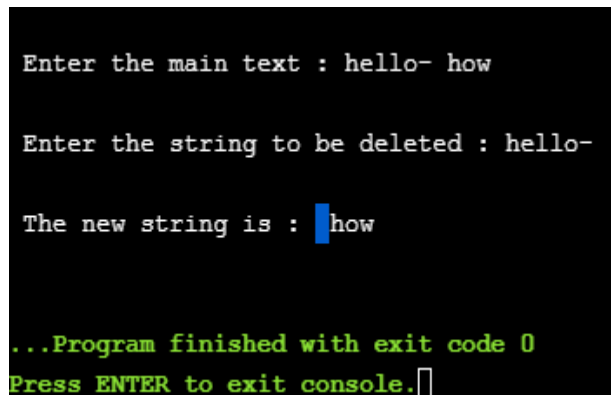
Algorithm to delete a substring from a text

Write a program to delete a substring from a text.

```
#include <stdio.h>

int main() {
    char text[200], str[20], new_text[200];
    int i=0, j=0, found=0, k, n=0, copy_loop=0;
    printf("\n Enter the main text : ");
    gets(text);
    printf("\n Enter the string to be deleted : ");
    gets(str);
    while(text[i]!='\0'){
        j=0, found=0, k=i;
        while(text[k]==str[j] && str[j]!='\0')
        {
            k++;
            j++;
        }
        if(str[j]=='\0')
            copy_loop=k;
        new_text[n] = text[copy_loop];
        i++;
        copy_loop++;
        n++;
    }
    new_text[n]='\0';
    printf("\n The new string is : ");
    puts(new_text);
    getch();
    return 0;
}
```

OUTPUT



```
Enter the main text : hello- how

Enter the string to be deleted : hello-

The new string is : how

...Program finished with exit code 0
Press ENTER to exit console.
```

x) Replacement

Replacing a Pattern with Another Pattern in a String

The replacement operation is used to replace the pattern P1 by another pattern P2 . This is done by writing REPLACE(text, pattern , pattern).

For example,

("AAABBBCCC", "BBB", "X") = AAAXCCC

("AAABBBCCC", "X", "YYY")= AAABBBCC

In the second example, there is no change as X does not appear in the text.

Figure shows an algorithm to replace a pattern P1 with another pattern P2 in the text.

The algorithm is very simple, where we first find the position POS, at which the pattern occurs in the text, then delete the existing pattern from that position and insert a new pattern there.

```
Step 1: [INITIALIZE] SET POS = INDEX(TEXT, P1)  
Step 2: SET TEXT = DELETE(TEXT, POS, LENGTH(P1))  
Step 3: INSERT(TEXT, POS, P2)  
Step 4: EXIT
```

Algorithm to replace a pattern P₁ with another pattern P₂ in the text

Write a program to replace a pattern with another pattern in the text.

```
#include <stdio.h>
int main() {
char str[200], pat[20], new_str[200], rep_pat[100]; int i=0, j=0, k, n=0,
    copy_loop=0, rep_index=0;
printf("\n Enter the string : ");
gets(str);
printf("\n Enter the pattern to be replaced: ");
gets(pat);
printf("\n Enter the replacing pattern: ");
gets(rep_pat);
while(str[i]!='\0'){
j=0,k=i;
while(str[k]==pat[j] && pat[j]!='\0'){
    k++;
    j++;}
if(pat[j]!='\0'){
    copy_loop=k;
    while(rep_pat[rep_index] !='\0'){
        new_str[n] = rep_pat[rep_index];
        rep_index++;
        n++;}}
new_str[n] = str[copy_loop];
i++;
copy_loop++;
n++;}
new_str[n]='\0';
printf("\n The new string is : ");
puts(new_str);
return 0;
}
```

OUTPUT

```
Enter the string : how ARE u?
Enter the pattern to be replaced: ARE
Enter the replacing pattern: are
The new string is : how are u?
```

ARRAY OF STRINGS

- ❁ Till now we have seen that a string is an array of characters. For example, if we say `char name[] = "Mohan"`, then the name is a string (character array) that has five characters.
- ❁ Now, suppose that there are 5 students in a class and we need a string that stores the names of all the 5 students. How can this be done? Here, we need a string of strings or an array of strings.
- ❁ Such an array of strings would store 5 individual strings. An array of strings is declared as `char names[5][10]`;
- ❁ Here, the first index will specify how many strings are needed and the second index will specify the length of every individual string. So here, we will allocate space for 5 names where each name can be a maximum 10 characters long.
- ❁ Let us see the memory representation of an array of strings. If we have an array declared as

```
char name[5][10] = {"Ram", "Mohan", "Shyam", "Hari", "Gopal"};
```

- ❁ Then in the memory, the array will be stored as shown in Fig.

name[0]	R	A	M	'\0'					
name[1]	M	O	H	A	N	'\0'			
name[2]	S	H	Y	A	M	'\0'			
name[3]	H	A	R	I	'\0'				
name[4]	G	O	P	A	L	'\0'			

Memory representation of a 2D character array

- ✿ By declaring the array names, we allocate 50 bytes. But the actual memory occupied is 27 bytes. Thus, we see that about half of the memory allocated is wasted.
- ✿ Figure shows an algorithm to process individual string from an array of strings.
- ✿ In Step 1, we initialize the index variable I to zero. In Step 2, a while loop is executed until all the strings in the array are accessed. In Step 3, each individual string is processed

```

Step 1: [INITIALIZE] SET I=0
Step 2: Repeat Step 3 while I< N
Step 3:     Apply Process to NAMES[I]
          [END OF LOOP]
Step 4: EXIT

```

Algorithm to process individual string from an array of strings

WRITE A PROGRAM TO READ AND PRINT THE NAMES OF N STUDENTS OF A CLASS

```

#include<stdio.h>
#include<conio.h>
main()
{
    char names[5][10];
    int i, n;
    clrscr();

    printf("\n Enter the number of students : ");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    {
        printf("\n Enter the name of %dth student : ", i+1);
        gets(names[i]);
    }
    printf("\n Names of the students are : \n");
    for(i=0;i<n;i++)
        puts(names[i]);
    getch();
    return 0;
}

```

```
#include<stdio.h>

int main(){
    char line[150];
    int i,j;
    printf("Enter a string: ");
    gets(line);
    for(i=0; line[i]!='\0'; ++i)
    {
        while (!((line[i]>='a'&&line[i]<='z') ||
(line[i]>='A'&&line[i]<='Z' || line[i]=='\0'))))
        {
            for(j=i;line[j]!='\0';++j)
            {
                line[j]=line[j+1];
            }
            line[j]='\0';
        }
    }

    printf("Output String: ");
    puts(line);
    return 0;
}
```

This program takes a string from user and *for* loop executed until all characters of string is checked. If any character inside a string is not a alphabet, all characters after it including null character is shifted by 1 position backwards.

Enter a string: p2'r"o@gram84iz./
Output String: programiz

Introduction to Pointers

Understanding The Computer's Memory

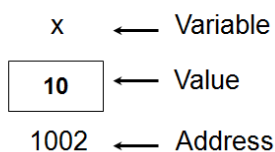
- ✿ Every computer has a primary memory. All our data and programs need to be placed in the primary memory for execution.
- ✿ The primary memory or RAM (Random Access Memory which is a part of the primary memory) is a collection of memory locations (often known as cells) and each location has a specific address. Each memory location is capable of storing 1 byte of data
- ✿ Generally, the computer has four areas of memory each of which is used for a specific task. These areas of memory include- stack, heap and global memory.
- ✿ **Stack-** A fixed size of stack is allocated by the system and is filled as needed from the bottom to the top, one element at a time. These elements can be removed from the top to the bottom by removing one element at a time. That is, the last element added to the stack is removed first.
- ✿ **Heap-** Heap is a contiguous block of memory that is available for use by the program when need arise. A fixed size heap is allocated by the system and is used by the system in a random fashion.
- ✿ When the program requests a block of memory, the dynamic allocation technique carves out a block from the heap and assigns it to the program.
- ✿ When the program has finished using that block, it returns that memory block to the heap and the location of the memory locations in that block is added to the free list.
- ✿ **Global Memory-** The block of code that is the main() program (along with other functions in the program) is stored in the global memory. The memory in the global area is allocated randomly to store the code of different functions in the program in such a way that one function is not contiguous to another function. Besides, the function code, all global variables declared in the program are stored in the global memory area.
- ✿ **Other Memory Layouts-** C provides some more memory areas like- text segment, BSS and shared library segment.
- ✿ The text segment is used to store the machine instructions corresponding to the compiled program. This is generally a read-only memory segment
- ✿ BSS is used to store un-initialized global variables
- ✿ Shared libraries segment contains the executable image of shared libraries that are being used by the program.

Intro to pointers

- ✿ Every variable in C has a name and a value associated with it. When a variable is declared, a specific block of memory within the computer is allocated to hold the value of that variable. The size of the allocated block depends on the type of the data.
`int x = 10;`
- ✿ When this statement executes, the compiler sets aside 2 bytes of memory to hold the value 10. It also sets up a symbol table in which it adds the symbol x and the relative address in memory where those 2 bytes were set aside.
- ✿ Thus, every variable in C has a value and an also a memory location (commonly known as address) associated with it. Some texts use the term rvalue and lvalue for the value and the address of the variable respectively.
- ✿ The rvalue appears on the right side of the assignment statement and cannot be used on the left side of the assignment statement. Therefore, writing `10 = x;` is illegal.

Example:

`x=10`



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

```
void main(){
```

```
    int x=10;
```

```
    printf("\n The Address of x = %u",&x);
```

```
    printf("\n The Value of x = %d",x);}
```

OUTPUT

The Address of x = 1002

The Value of x = 10


```

#include<conio.h>
#include<stdio.h>
int main()
{
    int *ptr1,*ptr2,a,b;
    clrscr();
    printf("Enter two numbers\n");
    scanf("%d%d",&a,&b);
    printf("Given numbers are %d and %d\n",a,b);
    ptr1=&a;
    ptr2=&b;
    printf("Address of a is %x and that of b is %x\n",ptr1,ptr2);
    printf("Sum of %d and %d is %d\n",a,b,*ptr1+*ptr2);
    getch();
    return 0;
}

```

Variable name ----->



Address ----->

Declaring Pointer Variables

- ✿ Actually pointers are nothing but memory addresses.
- ✿ A pointer is a variable that contains the memory location of another variable.
- ✿ The general syntax of declaring pointer variable is

`data_type *ptr_name;`

Here, data-type - Type of the data to which the pointer points.
 pointer-name - Name of the pointer

For example:

`int *pnum; char *pch; float *pfnum;`

Accessing Variable through Pointer

If a pointer is declared and assigned to a variable, then the variable can be accessed through the pointer.

Example:

```
int *ptr;  
int x= 10;  
ptr = &x;
```

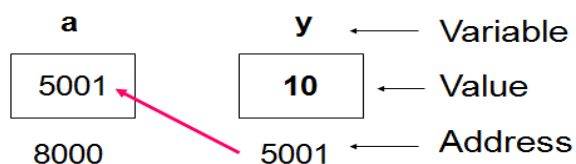
The '*' informs the compiler that ptr is a pointer variable and the int specifies that it will store the address of an integer variable.

The & operator retrieves the lvalue (address) of x, and copies that to the contents of the pointer ptr.

Example:

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
  int x=5;  
  int *a;  
  a=&x;  
  printf("\n The Value of x = %d",x);  
  printf("\n The Address of x = %u",&x);  
  printf("\n The Value of a = %d",a);  
  printf("\n The Value of x = %d",*a);  
}
```

The Value of x = 5
The Address of x = 8758
The Value of a = 8758
The Value of x = 5



Example:

```
#include<stdio.h>  
#include<conio.h>  
void main(){  
  int y=10;  
  int *a;  
  a=&y;  
  printf("\n The Value of y = %d",y);  
  printf("\n The Address of y = %u",&y);  
  printf("\n The Value of a = %d",a);  
  printf("\n The Address of a = %u",&a);}  
The Value of y = 10  
The Address of y = 5001  
The Value of a = 5001  
The Address of a = 8000
```

De-referencing A Pointer Variable

We can "dereference" a pointer, i.e. refer to the value of the variable to which it points by using unary '*' operator as in *ptr. That is, *ptr = 10, since 10 is value of x.

```
#include<stdio.h>
int main()
{
    int num, *pnum;
    pnum = &num;
    printf("\n Enter the number : ");
    scanf("%d", &num);
    printf("\n The number that was entered is : %d", *pnum);
    return 0;
}
```

OUTPUT:

Enter the number : 10

The number that was entered is : 10



```
char option = 'Y';
```

Allots some memory location 4042 (for example) with a name option and stores value 'Y' in it

Memory Address of variable 'option'

Value in 'option'

'Y'

option

Variable

4042

```
char *ptr = NULL;
```

Creates a pointer variable with a name 'ptr' Which can hold a Memory address

ptr

```
ptr = &option;
```

Memory address of Variable 'option' Is copied to the Pointer 'ptr'

4042

ptr

'Y'

option

4042

```
*ptr = 'N';
```

The value 'N' is stored in the variable which has the memory address 4042

4042

ptr

'N'

option

4042

```
int main() {
```

pointer variables are declared

```
int n1, n2 ;
```

```
int *p = "NULL", *q = "NULL";
```

```
n1 = 6 ;
```

```
p = &n1;
```

Prints 6 6

```
printf ("\n%d value1 %d", n1,*p);
```

Prints address of n1

```
printf ("\n%d value2 %d",&n1, p );
```

```
q = &n2;
```

```
*q = 3 ;
```

```
printf ("\n %d value3 %d ", *p , *q ) ;
```

Prints 6 3

```
p = q;
```

pointer 'q' assigned with pointer 'q'

```
printf ("\n %d value4 %d ", *p , *q ) ;
```

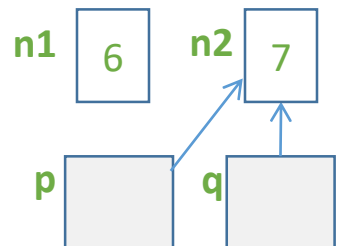
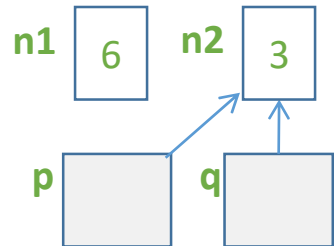
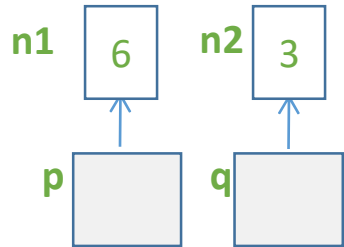
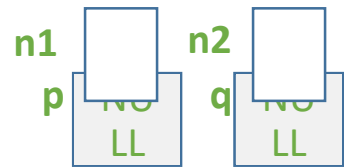
Prints 3 3

```
*p = 7 ;
```

```
printf ("\n %d value5 %d ", *p , *q ) ;
```

```
}
```

Prints 7 7



When two pointers are referencing with one variable, both pointers contains address of the same variable, and the value changed through with one pointer will reflect to both of them.

Pointer operators / expressions

- ✿ Pointer variables can also be used in expressions. For ex,

```
int num1=2, num2= 3, sum=0, mul=0, div=1;
```

```
int *ptr1, *ptr2;
```

```
ptr1 = &num1, ptr2 = &num2;
```

```
sum = *ptr1 + *ptr2;
```

```
mul = sum * *ptr1;
```

```
*ptr2 +=1;
```

```
div = 9 + *ptr1/*ptr2 - 30;
```

- ✿ We can add integers to or subtract integers from pointers as well as to subtract one pointer from the other.
- ✿ We can compare pointers by using relational operators in the expressions. For example $p1 > p2$, $p1==p2$ and $p1!=p2$ are all valid in C.

When using pointers, unary increment (++) and decrement (--) operators have greater precedence than the dereference operator (*).

Therefore, the expression

$*ptr++$ is equivalent to $*(ptr++)$.

So the expression will increase the value of ptr so that it now points to the next element.

In order to increment the value of the variable whose address is stored in ptr, write $(*ptr)++$

Pointer arithmetic and arrays

```
#include <stdio.h>
int main() {
    int arr [5] = { 12, 31, 56, 19, 42};
    int *p;
    p = arr + 1;
    printf("%d \n", *p);
    printf("%d %d %d\n", *(p-1), *(p), *(p + 1));
    --p;
    printf("%d", *p);
}
```

Prints 31

Prints 12 31 56

Prints 12

arr[0] or *(arr + 0) → 12 → p - 1

arr[1] or *(arr + 1) → 31 → p

arr[2] or *(arr + 2) → 56 → p + 1

arr[3] or *(arr + 3) → 19 → p + 2

arr[4] or *(arr + 4) → 42 → p + 3

Subscript operator [] used to access an element of array implements address arithmetic, like pointer.

```

#include<stdio.h>
#include<conio.h>
int main()
{
    int i,n,m[10],*ptr=m;
    clrscr();
    printf("Enter the size of array\n");
    scanf("%d",&n);
    printf("enter %d numbers\n",n);
    for(i=0;i<n;i++)
        scanf("%d",&m[i]);
    for(i=0;i<n;i++)
    {
        printf("<ptr+%d>=%x\t",i,<ptr+i>);
        printf("*<ptr+%d>=%d\n",i,*<ptr+i>);
    }
    getch();
    return 0;
}

```

Array index ----->

			ARRAY				
--	--	--	-------	--	--	--	--

Address----->

Null Pointers

- 1) A *null pointer* which is a special pointer value that is known not to point anywhere. This means that a NULL pointer does not point to any valid memory address.

To declare a null pointer you may use the predefined constant NULL,

```
int *ptr = NULL;
```

You can always check whether a given pointer variable stores address of some variable or contains a null by writing,

```
if ( ptr == NULL)
{
    Statement block;
}
```

Null pointers are used in situations if one of the pointers in the program points somewhere some of the time but not all of the time. In such situations it is always better to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it.

- 2) A pointer is said to be null pointer if zero is assigned to the pointer.

Example

```
int *a,*b;
a=b=0;
```

Generic Pointers / void pointers

- ✿ A generic pointer is pointer variable that has void as its data type.
- ✿ The generic pointer, can be pointed at variables of any data type.
- ✿ It is declared by writing

```
void *ptr;
```
- ✿ You need to cast a void pointer to another kind of pointer before using it.
- ✿ Generic pointers are used when a pointer has to point to data of different types at different times.

For example 1 ,

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

```
int main()
```

```
{ int x=10;
    char ch = 'A';
    void *gp;
    gp = &x;
    printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
    gp = &ch;
    printf("\n Generic pointer now points to the character %c", *(char*)gp);
    return 0;
}
```

OUTPUT:

Generic pointer points to the integer value = 10

Generic pointer now points to the character = A

For example 2 ,

'void' type pointer is a generic pointer, which can be assigned to any data type without cast during compilation or runtime. 'void' pointer cannot be dereferenced unless it is cast.

```
int main( ) {
    void* p;
    int x = 7;
    float y = 23.5;
    p = &x;
    printf("x contains : %d\n", *( ( int *)p) );
    p = &y;
    printf("y contains : %f\n", *( ( float *)p) );
}
```

OUTPUT

x contains 7

y contains 23.500000

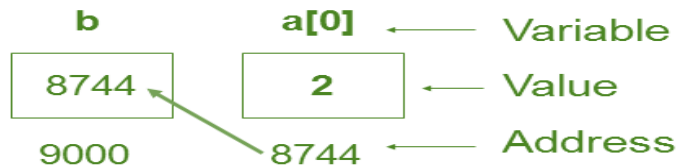
Pointers And Arrays

Pointers And One Dimensional Array

✿ The elements of the array can also be accessed through a pointer.

✿ Example

```
int a[3]={2,3,7};  
int *b;  
b=a;
```



Example:

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int a[3]={2,3,7};  
    int *b;  
    b=a;  
    printf("\n The Value of a[0] = %d",a[0]);  
    printf("\n The Address of a[0] = %u",&a[0]);  
    printf("\n The Value of b = %d",b);  
}
```

OUTPUT

```
The Value of a[0] = 2  
The Address of a[0] = 8744  
The Value of b = 8744
```

Guided Activity : Pointer and Arrays

Even though pointers and arrays work alike and strongly related, they are not synonymous. When an array is assigned with pointer, the address of first element of the array is copied into the pointer.

```
#include<stdio.h>
int main()
{
    int a[3] = { 12, 5 ,7}, b[3];
    int *p,*q;
```

Prints 12 12



```
    p = a;
    printf("%d %d\n", *p, *a);
```

Prints 12 12



```
    q = p;
    printf("%d %d",*p,*q);
```

```
    b = a; /* error */
}
```

Pointer is an address variable, having no initialized value by default. The address stored in the pointer can be changed time to time in the program.

Array name is an address constant, initialized with the address of the first element (base address)in the array. The address stored in array name cannot be changed in the program.

Pointers And Two Dimensional Array

a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2] a[2][0] a[2][1] a[2][2] a[3][0] a[3][1] a[3][2]



base_address

Array name contains base address

Address of $a[i][j] = *(*(\text{base_address} + i) + j) = *((a + i) + j)$

- Individual elements of the array mat can be accessed using either:
mat[i][j] or $*(*(\text{mat} + i) + j)$ or $*(\text{mat}[i] + j)$;
- See pointer to a one dimensional array can be declared as,
int arr[]={1,2,3,4,5};
int *parr;
parr=arr;
- Similarly, pointer to a two dimensional array can be declared as,
int arr[2][2]={{1,2},{3,4}};
int (*parr)[2];
parr=arr;
- Look at the code given below which illustrates the use of a pointer to a two dimensional array.

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

```
main()
```

```
{
    int arr[2][2]={{1,2},{3,4}};
    int i, (*parr)[2];
    parr=arr;
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
            printf(" %d", (*(parr+i))[j]);
    }
}
```

OUTPUT

1 2 3 4

Pointers And Strings

✿ Now, consider the following program that prints a text.

```
#include<stdio.h>

main()

{ char str[] = "Oxford";
    char *pstr = str;

    printf("\n The string is : ");
    while( *pstr != '\0')
    {      printf("%c', *pstr);
          pstr++;
    }
}
```

✿ In this program we declare a character pointer *pstr to show the string on the screen. We then "point" the pointer pstr at str. Then we print each character of the string in the while loop. Instead of using the while loop, we could have straight away used the function puts(), like puts(pstr);

✿ Consider here that the function prototype for puts() is:

✿ int puts(const char *s); Here the "const" modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer. Note that the address of the string is passed to the function as an argument.

Array Of Pointers

An array of pointers can be declared as

```
int *ptr[10]
```

The above statement declares an array of 10 pointers where each of the pointer points to an integer variable. For example, look at the code given below.

```
int *ptr[10];  
  
int p=1, q=2, r=3, s=4, t=5;  
  
ptr[0]=&p;  
ptr[1]=&q;  
ptr[2]=&r;  
ptr[3]=&s;  
ptr[4]=&t
```

Can you tell what will be the output of the following statement?

```
printf("\n %d", *ptr[3]);
```

Yes, the output will be 4 because ptr[3] stores the address of integer variable s and *ptr[3] will therefore print the value of s that is 4.

- ✿ The advantage of pointer array is that the length of each row in the array may be different. The important application of pointer array is to store character strings of different length. Example :
- ✿

```
char *day[ ] = { "Sunday", "Monday", "Tuesday", "Wednesday",  
Thursday", "Friday", "Saturday" };
```

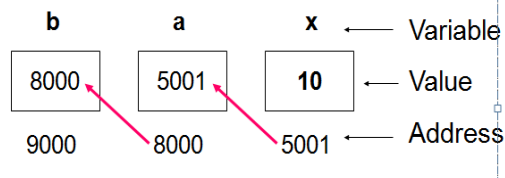
Pointers To Pointers (Double Indirection)

- ❁ You can use pointers that point to pointers. The pointers in turn, point to data (or even to other pointers). To declare pointers to pointers just add an asterisk (*) for each level of reference.

Here one pointer stores the address of another pointer variable.

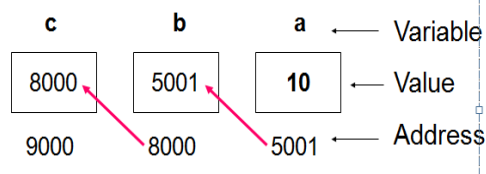
- ❁ Example:

```
int x=10,*a,**b;  
a=&x;  
b=&a;
```



Example

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int a=10;  
    int *b,**c;  
    b=&a;  
    c=&b;  
    printf("\n The Value of a = %d",a);  
    printf("\n The Address of a = %u",&a);  
    printf("\n The Value of b = %d",b);  
    printf("\n The Address of b = %u",&b);  
    printf("\n The Value of c = %d",c);  
    printf("\n The Address of c = %u",&c);  
}
```



OUTPUT

```
The Value of a = 10  
The Address of a = 5001  
The Value of b = 5001  
The Address of b = 8000  
The Value of c = 8000  
The Address of c = 9000
```


Drawbacks of Pointers

Although pointers are very useful in C, they are not free from limitations. If used incorrectly, pointers can lead to bugs that are difficult to unearth. For example, if you use a pointer to read a memory location but that pointer is pointing to an incorrect location, then you may end up reading a wrong value. An erroneous input always leads to an erroneous output. Thus however efficient your program code may be, the output will always be disastrous. Same is the case when writing a value to a particular memory location.

Let us try to find some common errors when using pointers.

```
int x, *px;  
x=10;  
*px = 20;
```

Error: Un-initialized pointer. px is pointing to an unknown memory location. Hence it will overwrite that location's contents and store 20 in it.

```
int x, *px;  
x=10;  
px = x;
```

Error: It should be `px = &x;`

```
int x=10, y=20, *px, *py; px = &x, py = &y; if(px<py)  
printf("\n x is less than y"); else  
printf("\n y is less than x");
```

Error: It should be `if(*px< *py)`

Exercise Programs

Exercise Programs:

Exercise program1 : To find the frequency of a character in a string

- ✿ we will learn **how to find occurrence of a particular character in a string using C program?**
- ✿ Here, we are reading a character array/string (character array is declaring with the maximum number of character using a Macro MAX that means maximum number of characters in a string should not more than MAX (100), then we are reading a character to find the occurrence and counting the characters which are equal to input character.
- ✿ **For example:**
If input string is **"Hello world!"** and we want to find occurrence of **'l'** in the string, output will be **'l' found 3 times in "Hello world!"**.

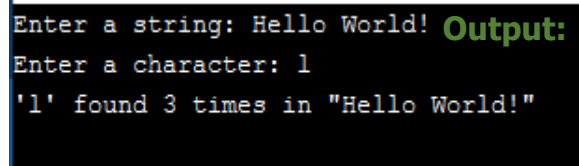
```
#include <stdio.h>
#define MAX    100
int main(){
char str[MAX]={0};
char ch;
int count,i;

//input a string
printf("Enter a string: ");
scanf("%[^\\n]s",str); //read string with spaces

getchar(); // get extra character (enter/return key)

//input character to check frequency
printf("Enter a character: ");
ch=getchar();

//calculate frequency of character
count=0;
for(i=0; str[i]!='\\0'; i++)
{
    if(str[i]==ch)
        count++;
}
printf("%c found %d times in %s",ch,count,str);
return 0;}
```

A screenshot of a terminal window showing the output of the C program. The text is as follows:
Enter a string: Hello World! **Output:**
Enter a character: l
'l' found 3 times in "Hello World!"

```
Enter a string: Hello World! Output:
Enter a character: l
'l' found 3 times in "Hello World!"
```

To find the frequency of all character s in a string

This program counts the frequency of characters in a string, i.e., which character is present how many times in the string. For example, in the string "code" each of the characters 'c,' 'd,' 'e,' and 'o' has occurred one time. Only **lower case alphabets** are considered, other characters (uppercase and special characters) are ignored. You can modify this program to handle uppercase and special symbols.

Explanation of "count[string[c]-'a']++", suppose input string begins with 'a' so c is 0 initially and string[0] = 'a' and string[0] - 'a' = 0 and we increment count[0], i.e., 'a' has occurred one time and repeat this until the complete string is scanned.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char string[100];
    int c = 0, count[26] = {0}, x;
    printf("Enter a string\n");
    gets(string);
    while (string[c] != '\0') {
        /* Considering characters from 'a' to 'z' only and ignoring others. */
        if (string[c] >= 'a' && string[c] <= 'z') {
            x = string[c] - 'a';
            count[x]++;
        }
        c++;
    }
    for (c = 0; c < 26; c++)
        printf("%c occurs %d times in the string.\n", c + 'a', count[c]);
    return 0;
}
```

OUTPUT

```
Enter a string
This program is easy 2 understand
a occurs 3 times in the string.
b occurs 0 times in the string.
c occurs 0 times in the string.
d occurs 2 times in the string.
e occurs 2 times in the string.
f occurs 0 times in the string.
g occurs 1 times in the string.
```

Exercise program 2: To find the number of vowels, consonants and white spaces in a given text

```
#include <stdio.h>

int main() {
    char line[150];
    int vowels, consonant, digit, space;
    vowels = consonant = digit = space = 0;
    printf("Enter a line of string: ");
    gets(line);
    //fgets(line, sizeof(line), stdin);
    for (int i = 0; line[i] != '\0'; ++i) {
        if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' ||
            line[i] == 'o' || line[i] == 'u' || line[i] == 'A' ||
            line[i] == 'E' || line[i] == 'I' || line[i] == 'O' ||
            line[i] == 'U') {
            ++vowels;
        } else if ((line[i] >= 'a' && line[i] <= 'z') || (line[i] >= 'A' && line[i] <= 'Z')) {
            ++consonant;
        } else if (line[i] >= '0' && line[i] <= '9') {
            ++digit;
        } else if (line[i] == ' ') {
            ++space;
        }
    }
    printf("Vowels: %d", vowels);
    printf("\nConsonants: %d", consonant);
    printf("\nDigits: %d", digit);
    printf("\nWhite spaces: %d", space);
    return 0;}
```

OUTPUT

```
main.c:(.text+0x62): warning: the 'gets' function is dangerous and should not be used.
Enter a line of string: This program is easy 2 understand
Vowels: 9
Consonants: 18
Digits: 1
White spaces: 5

...Program finished with exit code 0
Press ENTER to exit console.[]
```

```
/* C Program to Sort array of nos ascending and descending order */
```

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

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

```
void main()
```

```
{
```

```
    int a[10], t ;
```

```
    int i, j, n;
```

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

```
    scanf("%d", &n);
```

```
    printf("Enter %d numbers: \n" , n);
```

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

```
    {
```

```
        scanf("%d", &a[i]);
```

```
    }
```

```
    for (i = 0; i < n - 1 ; i++)
```

```
    {
```

```
        for (j = i + 1; j < n; j++)
```

```
        {
```

```
            if (a[i]>a[j])
```

```
            {
```

```
                t=a[i];
```

```
                a[i]=a[j];
```

```
                a[j]=t;
```

```
            }
```

```
        }
```

```
    }
```

```
    printf("Ascending Order Nos\n");
```

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

```
    {
```

```
        printf("%d\t", a[i]);
```

```
    }
```

```
    printf("\nDescending Order Nos\n");
```

```
    for (i = n-1; i >= 0; i--)
```

```
    {
```

```
        printf("%d\t", a[i]);
```

```
    }
```

```
}
```

OUTPUT

```
Enter the value of n
5
Enter 5 numbers:
7 3 5 4 1
Ascending Order Nos
1      3      4      5      7
Descending Order Nos
7      5      4      3      1

...Program finished with exit code 2
Press ENTER to exit console.
```

Exercise program3: Sorting the names

/* C Program to Sort Names in Alphabetical Order */

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

void main()
{
    char name[10][8], tname[10][8], temp[8];
    int i, j, n;

    printf("Enter the value of n \n");
    scanf("%d", &n);
    printf("Enter %d names: \n", n);
    for (i = 0; i < n; i++)
    {
        scanf("%s", name[i]);
        strcpy(tname[i], name[i]);
    }
    for (i = 0; i < n - 1; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (strcmp(name[i], name[j]) > 0)
            {
                strcpy(temp, name[i]);
                strcpy(name[i], name[j]);
                strcpy(name[j], temp);
            }
        }
    }
    printf("\n_____ \n");
    printf("Input NamestSorted names\n");
    printf("_____ \n");
    for (i = 0; i < n; i++)
    {
        printf("%s\t\t%s\n", tname[i], name[i]);
    }
    printf("_____ \n");
}
```

OUTPUT

```
Enter the value of n
3
Enter 3 names:
xerox
apple
Apple

-----
Input NamestSorted names
-----
xerox          Apple
apple          apple
Apple          xerox
-----
```

Assignment

Unit III

Assignment Questions

CO 1	Develop C program solutions to simple computational problems		
1.	Write a program in C that removes leading and trailing spaces from a string.	K2	CO4
2.	Write a program in C that replaces a given character with another character in a string.	K2	CO4

Part A

Question & Answer

Part A

1. What is the difference between „a“ and “a”?

„a“ is a character constant and “a” is a string.

2. What is the use of „\0“ character?

When declaring character arrays (strings), „\0“ (NULL) character is automatically added at end. The „\0“ character acts as an end of character array.

3. Define Strings.

The group of characters, digit and symbols enclosed within quotes is called as String (or) character Arrays. Strings are always terminated with „\0“ (NULL) character. The compiler automatically adds „\0“ at the end of the strings.

Example for character arrays [strings].

```
#include <stdio.h> main()
{
static char name1[] = {'H','e','l','l','o'}; static char name2[] = "Hello";
printf("%s\n", name1);
printf("%s\n", name2);}
```

4. List the different methods for reading and writing a string.

The different methods for reading a string are,

- scanf()
- gets()
- getchar()

getch() or getche()

5. The different methods for writing a string are,

printf()

puts()

putchar()

6. Write a C program to get a string input and print it.

```
#include<stdio.h> #include<conio.h> void main()
```

```
{      Output:
```

```
Excellent
```

```
char str[20];    The given string
```

```
Excellent
```

```
gets(str);
```

```
printf("The given string\n"); printf("%s",str);
```

```
}
```

7. What is the use of gets() function?

The gets() function allows a full line entry from the user. When the user presses the enter key to end the input, the entire line of characters is stored to a string variable.

8. Write a C program to find the length of given string.

```
#include <stdio.h> int main()
```

```
{
```

```
char s[1000], i;
```

Output:

```
printf("Enter a string: ");
```

Enter a string: hai

Programming in C

```
scanf("%s", s);
```

Length of string:16

```
for(i = 0; s[i] != '\0'; ++i); printf("Length of string: %d", i); return 0;
```

```
}
```

9. Write a C program to get a string input and print it.

```
#include<st
```

```
dio.h>
```

```
#include<co
```

```
nio.h> void
```

```
main()
```

```
{
```

Output:

Excellent

```
char str[20];
```

The given string

Excellent

```
gets(str);
```

```
printf("The given
```

```
string\n");
```

```
printf("%s",str);
```

```
}
```

10. Why is it necessary to give the size of an array in an array declaration?

When an array is declared, the compiler allocates a base address and reserves enough space in the memory for all the elements of the array. The size is required to allocate the required space. Thus, the size must be mentioned.

11. What is the use of gets() function?

The gets() function allows a full line entry from the user. When the user presses the enter key to end the input, the entire line of characters is stored to a string variable.

12. What is pointer?

Every variable in C has a name and a value associated with it. When a variable is declared, a specific block of memory within the computer is allocated to hold the value of that variable. The size of the allocated block depends on the data type. A pointer is a variable that contains the address of another variable.

13. How to declaring pointer variables?

The general syntax of declaring pointer variables can be given as below.

```
data_type *ptr_name;
```

14. Define pointer to pointer.

The pointers in turn point to data or even to other pointers. To declare pointer to pointer, just add an asterisk * for each level of reference

15. Write the drawbacks of pointers.

Pointers are very useful in C, they are not free from limitations. If used incorrectly, pointers can lead to bugs that are difficult to unearth.

Example: If you use a pointer to read a memory location but that pointer is pointing to an incorrect location, then you may end up reading a wrong value. An erroneous input always leads to an erroneous output. Thus however efficient our program code may be, the output will always be disastrous.

16. Differentiate * and &.

*	Value at operator	Gives value stored at particular address
&	Address operator	Gives address of variable

Part B

Questions

Part B

1. Write a C program to count number of words in a sentence.
2. Write a C program to check whether a string is palindrome.
3. Explain the standard string functions with example to support each type.
4. Explain how strings can be displayed on the screen.
5. Explain the syntax of printf() and scanf().
6. List all the substrings that can be formed from the string 'ABCD'.
7. What do you understand by pattern matching? Give an algorithm for it.
8. Write a short note on array of strings.
9. Explain with an example how an array of strings is stored in the main memory.
10. Explain how pointers and strings are related to each other with the help of a suitable program.
11. If the substring function is given as SUBSTRING (string, position, length), then find S(5, 9) if S = "Welcome to World of C Programming"
12. If the index function is given as INDEX(text, pattern), then find index(T, P) where T = "Welcome to World of C Programming" and P = "of"
12. Differentiate between gets() and scanf().
13. Give the drawbacks of getchar() and scanf().
14. Which function can be used to overcome the shortcomings of getchar() and scanf()?
15. How can putchar() be used to print a string?
16. Differentiate between a character and a string.
17. Differentiate between a character array and a string.
18. Write a program in which a string is passed as an argument to a function.
19. Write a program in C to concatenate first n characters of a string with another string.
20. Write a program in C that compares first n characters of one string with first n characters of another string.
- 21. Explain in detail about pointers with example.**
22. Write a C program to access array elements using Pointers
- 23. Write a C program to sort a list in alphabetic order using pointers.**

Supportive Online Certification

Unit III

Certification Courses

⚙ NPTEL

Problem solving through Programming in C

<https://nptel.ac.in/courses/106/105/106105171/>

⚙ Coursera

1) C for Everyone: Structured Programming

<https://www.coursera.org/learn/c-structured-programming>

2) C for Everyone: Programming Fundamentals

<https://www.coursera.org/learn/c-for-everyone>

Real time Applications

Unit III

1) Phonebook application

a small phonebook code challenge to build the shortest : Functions include add contact, remove contact, search contact and display contacts.

Content beyond syllabus

Unit III

Content beyond syllabus

- 1) Character manipulation functions in c using Ctype.h library file

Assessment Schedule

Unit III

Prescribed Text book & References

Unit III

Text books & References

TEXT BOOK

1. Reema Thareja, "Programming in C", Oxford University Press, Second Edition, 2016

REFERENCES

1. Kernighan, B.W and Ritchie,D.M, "The C Programming language", Second Edition, Pearson Education, 2006
2. Paul Deitel and Harvey Deitel, "C How to Program", Seventh edition, Pearson Publication
3. Juneja, B. L and Anita Seth, "Programming in C", CENGAGE Learning India pvt. Ltd., 2011
4. Pradip Dey, Manas Ghosh, "Fundamentals of Computing and Programming in C", First Edition, Oxford University Press, 2009
5. ER and ETA , "CC Foundation Program Reference materials" Infosys Ltd.

Mini Project Suggestions

Unit III

- 1) Contact Management System
- 2) Personal diary Management system

Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

OCS752

INTRODUCTION TO C

PROGRAMMING

Department: : Electrical and Electronics Engineering

Batch/Year: 2017-2021

Created by: Dr. S. Meenakshi and A.S. Vibith

Date: 21-08-2020

Table of Contents

- ✿ Course Objectives
- ✿ Syllabus
- ✿ Course Outcomes (Cos)
- ✿ CO-PO Mapping
- ✿ Lecture Plan
- ✿ Activity based learning
- ✿ Lecture notes
- ✿ Assignments
- ✿ Part A Q&A
- ✿ Part B Qs
- ✿ List of Supportive online Certification courses
- ✿ Real time applications in day to day life and to industry
- ✿ Contents beyond Syllabus
- ✿ Assessment Schedule (proposed and actual date)
- ✿ Prescribed Text Books & Reference Books
- ✿ Mini Project Suggestions

Course Objectives

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C

3 0 0 3

OBJECTIVES

- ✿ To develop C Programs using basic programming constructs
- ✿ To develop C programs using arrays and strings
- ✿ To develop applications in C using functions and structures

Syllabus

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C 3 0 0 3

UNIT I INTRODUCTION

9

Structure of C program – Basics: Data Types – Constants – Variables - Keywords – Operators: Precedence and Associativity - Expressions - Input/output statements, Assignment statements – Decision-making statements - Switch statement - Looping statements – Pre-processor directives - Compilation process – Exercise Programs: Check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers

UNIT II ARRAYS

9

Introduction to Arrays – One dimensional arrays: Declaration – Initialization - Accessing elements – Operations: Traversal, Insertion, Deletion, Searching - Two dimensional arrays: Declaration – Initialization - Accessing elements – Operations: Read – Print – Sum – Transpose – Exercise Programs: Print the number of positive and negative values present in the array – Sort the numbers using bubble sort - Find whether the given is matrix is diagonal or not.

UNIT III STRINGS

9

Introduction to Strings - Reading and writing a string - String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse – Substring – Insertion – Indexing – Deletion – Replacement – Array of strings – Introduction to Pointers – Pointer operators – Pointer arithmetic - Exercise programs: To find the frequency of a character in a string - To find the number of vowels, consonants and white spaces in a given text - Sorting the names.

UNIT IV FUNCTIONS

9

Introduction to Functions – Types: User-defined and built-in functions - Function prototype – Function definition - Function call - Parameter passing: Pass by value - Pass by reference - Built-in functions (string functions) – Recursive functions – Exercise programs: Calculate the total amount of power consumed by 'n' devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)

UNIT V STRUCTURES

9

Introduction to structures – Declaration – Initialization – Accessing the members – Nested Structures – Array of Structures – Structures and functions – Passing an entire structure – Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

TOTAL:45 PERIODS

Course Outcomes

- ✿ CO 1 - Develop algorithmic solutions to simple computational problems using basic constructs K1
- ✿ CO 2 - Develop simple applications in C using Control Constructs K2
- ✿ CO 3 - Design and implement applications using arrays K2
- ✿ CO 4 – Represent data using string and string operations K3
- ✿ CO 5 - Decompose a C program into functions and pointers K3
- ✿ CO 6 - Represent and write program using structure and union K3

CO – PO Mapping

CO	PO	Mapping Level	Justification
CO	PO	Mapping Level	Justification
CO1	PO1	2	Identify the data type and operators to solve the problem
CO1	PO2	2	Design the expression in an efficient way
CO1	PO3	2	Recognize the need of basic c Tokens-variables-constants
CO1	PO5	2	Apply the concept of control statements for simple solving the problem
CO1	PO12	1	Formulate the iterative statements for problem solving
CO2	PO1	3	Develop a complete program s for preprocessor directives
CO2	PO2	3	Recognize the implementation of simple problem solving with above concepts
CO3	PO3	2	Apply simple mathematical concepts for writing 1D arrays and its operations
CO3	PO5	2	Identify and formulate for the given problem using 2D and its operations
CO3	PO12	1	Design way of problem solving in Multi Dimensional arrays
CO4	PO1	3	Recognize the need of implementation in string
CO4	PO2	3	Apply logic to solve simple problem statement using string operations
CO4	PO3	3	Apply the knowledge to find the possible code for string manipulations
CO5	PO5	2	Identify the code for decomposition as function
CO5	PO12	1	Develop functions and reuse it whenever required to reduce the lines of code
CO5	PO1	2	Recognize the need of function concepts
CO5	PO2	2	Apply compound data knowledge to select any one
CO5	PO3	2	Apply the concept of pointers
CO5	PO5	2	Design and Develop program using the selected compound data
CO6	PO12	1	Recognize the need of structure
CO6	PO1	2	Apply the basic idea of handling with union
CO6	PO2	2	Identify the number of modes and operations on structure in detail
CO6	PO3	2	Develop programs using structure and union

Lecture Plan

Unit IV

CO-PO/PSO MAPPING

COURSE OUTCOME	LEVEL OF COURSE OUTCOME	PROGRAM OUTCOME (PO)												PROGRAM SPECIFIC OUTCOME (PSO)			
		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	K1	2	2	2	-	2	-	-	-	-	-	-	1	1			
CO2	K2	3	3	2	-	2	-	-	-	-	-	-	1	1			
CO3	K2	3	3	3	-	2	-	-	-	-	-	-	1	1			
CO4	K3	2	2	2	-	2	-	-	-	-	-	-	1	1			
CO5	K3	2	2	2	-	-	-	-	-	-	-	-	1	1			
CO6	K3	3	3	3		2							1	1			

Unit IV - FUNCTIONS

S.No	Topics	No. of Periods	Proposed Date	Actual Lecture Date	Pertaining CO	Taxonomy Level	Mode of Delivery
1	Introduction to Functions	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
2	Types: User-defined and built-in functions	1			CO1	K2	PPT, Chalk & Talk
3	Function prototype – Function definition - Function call	1			CO1	K2	PPT, Chalk & Talk
4	Parameter passing: Pass by value - Pass by reference	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
5.6	Built-in functions (string functions) – Recursive functions	2			CO1	K2	PPT, Chalk & Talk
7	Exercise programs 1: Calculate the total amount of power consumed by 'n' devices (passing an array to a function)	1			CO1	K2	PPT, Chalk & Talk
8,9	Exercise Programs: Ex. Prog. 2 : Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)	2			CO1	K2	PPT, Chalk & Talk

Activity Based Learning

Unit IV

Activity Based Learning

- ✿ Learn by solving problems – Tutorial Sessions can be conducted
 - Tutorial sessions available in Skillrack for practice
- ✿ Learn by questioning
- ✿ Learn by doing hands-on IN ONLINR / VIRTUAL LAB.

Lecture Notes

UNIT IV FUNCTIONS

9

Introduction to Functions – Types: User-defined and built-in functions - Function prototype – Function definition - Function call - Parameter passing: Pass by value - Pass by reference - Built-in functions (string functions) – Recursive functions – Exercise programs: Calculate the total amount of power consumed by 'n' devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)

Unit IV - Functions : LEARNING PLAN

Sl. No.	Topics	Learning Content (hh.mm)	Post-Session (Quiz + Assignment) (hh:mm)
4.0 and 4.1	Introduction to Functions – Types: User-defined and built-in functions - Function prototype – Function definition	3.00	1.00
4.2	Function call - Parameter passing: Pass by value - Pass by reference - Built-in functions (string functions) – Recursive functions.	3.00	0.50
4.3	Exercise programs: Calculate the total amount of power consumed by 'n' devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)	3.00	1.00
	Total	9.00	2.50

Introduction to Functions

Introduction to Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

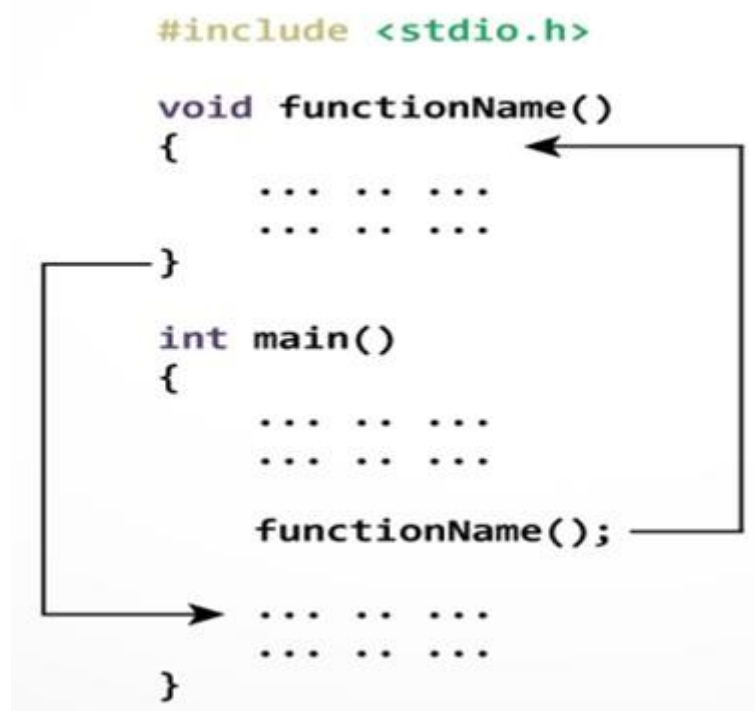
Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

Concept of Modularization

One of the most important concepts of programming is the ability to group some lines of code into a unit that can be included in our program. The original wording for this was a sub-program. Other names include: macro, sub-routine, procedure, module and function. We are going to use the term **function** for that is what they are called in most of the predominant programming languages of today. Functions are important because they allow us to take large complicated programs and to divide them into smaller manageable pieces. Because the function is a smaller piece of the overall program, we can concentrate on what we want it to do and test it to make sure it works properly. Generally, functions fall into two categories:

1. Program Control – Functions used to simply sub-divide and control the program. These functions are unique to the program being written. Other programs may use similar functions, maybe even functions with the same name, but the content of the functions are almost always very different.

2. Specific Task – Functions designed to be used with several programs. These functions perform a specific task and thus are usable in many different programs because the other programs also need to do the specific task. Specific task functions are sometimes referred to as building blocks. Because they are already coded and tested, we can use them with confidence to more efficiently write a large program.



Why we need functions in C

Functions are used because of following reasons

- a) To improve the readability of code.
- b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch
- c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.
- d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Terminologies of Functions

- A function `f` that uses another function `g`, is known as the calling function and `g` is known as the called function
- The input that the function takes are known as argument / parameter.
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass parameters to the called function. If the called function accepts arguments, the calling function will pass parameters, else it will not do so.

Scope of a Function:

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

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

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

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

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

Global Variables

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

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


```
#include <stdio.h>

/* global variable declaration */
int x;

int main () {
    /* local variable declaration */
    int a, b;

    /* actual initialization */
    a = 20;
    b = 10;

    x = a + b;

    printf ("value of a = %d, b = %d and x = %d\n", a, b, x);

    return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference.

Example

```
#include <stdio.h>

/* global variable declaration */
int x = 30;

int main ()
{
    /* local variable declaration */
    int x = 10;

    printf ("value of x = %d\n", x);

    return 0;
}
```

Types of Functions

Types of function

There are two types of function in C programming:

- **Standard library functions**
- **User-defined function**

Standard library functions

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

- The `printf()` is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the `stdio.h` header file. Hence, to use the `printf()` function, we need to include the `stdio.h` header file using `#include <stdio.h>`.

The `sqrt()` function calculates the square root of a number. The function is defined in the `math.h` header file.

Example:

Square root using `sqrt()` function

```
#include<stdio.h>
#include<math.h>
void main()
{
float number, root;
printf("Enter the number");
scanf("%f",&number);
root=sqrt(number);
printf("Square root of %.2f=%.2f",number,root);
}
```

Output

Enter the number 12
Square root of 12.00 = 3.46

Library Functions in Different Header Files

C Header Files	Purpose
<assert.h>	Program assertion functions
<ctype.h>	Character type functions
<locale.h>	Localization functions
<math.h>	Mathematics functions
<setjmp.h>	Jump functions
<signal.h>	Signal handling functions
<stdarg.h>	Variable arguments handling functions
<stdio.h>	Standard Input/Output functions
<stdlib.h>	Standard Utility functions
<string.h>	String handling functions
<time.h>	Date time functions

Advantages of Using C library functions

1. The functions are optimized for performance

Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.

2. It saves considerable development time

Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

3. The functions are portable

With ever-changing real-world needs, your application is expected to work every time, everywhere. And, these library functions help you in that they do the same thing on every computer.

User-defined function

The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.

Syntax

```
return_type function_name(argument list)
{
    set of statement block of code
}
```

Example:

```
// Creating a user defined function addition()

#include <stdio.h>

int addition(int number1, int number2)
{
    int sum;

    /* Arguments are used here*/
    sum = number1+number2;

    /* Function return type is integer so we are returning
    * an integer value, the sum of the passed numbers.
    */
    return sum;
}
```

```
int main()
{
    int variable1, variable2;
    printf("Enter number 1: ");
    scanf("%d",&variable1);
    printf("Enter number 2: ");
    scanf("%d",&variable2);

    /* Calling the function here, the function return type
    * is integer so we need an integer variable to hold the
    * returned value of this function.
    */
    int res = addition(variable1, variable2);
    printf ("Output: %d", res);
    return 0;
}
```

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

Function Prototype

Function Prototype:

The function prototypes are used to tell the compiler about the number of arguments and about the required datatypes of a function parameter, it also tells about the return type of the function. By this information, the compiler cross-checks the function signatures before calling it. If the function prototypes are not mentioned, then the program may be compiled with some warnings, and sometimes generate some strange output.

If some function is called somewhere, but its body is not defined yet, that is defined after the current line, then it may generate problems. The compiler does not find what is the function and what is its signature. In that case, we need to function prototypes. If the function is defined before then we do not need prototypes.

Example:

Without using prototype

```
#include<stdio.h>

main() {
    function(50);
}

void function(int x) {
    printf("The value of x is: %d", x);
}
```

Output:

The value of X is : 50

This shows the output, but it is showing some warning like below

[Warning] conflicting types for 'function'

[Note] previous implicit declaration of 'function' was here :

Example:

Using Prototype:

```
#include<stdio.h>

void function(int); //prototype

main() {
    function(50);
}

void function(int x) {
    printf("The value of x is: %d", x);
}
```

Output:

The value of x is: 50

Function Definition

Function Definition:

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function –

- **Return Type** – A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body – The function body contains a collection of statements that define what the function does.

Syntax:

```
Return_data_type function_name(data_type variable1, data_type variable2,...)
```

```
{  
    .....  
    statements  
    .....  
    return (variable);  
}
```

The number of arguments and the order of arguments in the function header must be same as that given in the function declaration statement.

While `return data_type function_name(data_type variable1, data_type variable2,...)` is known as the function header, the rest of the portion comprising of program statements within `{ }` is the function body which contains the code to perform the specific task.

The function header is same as that of function declaration. The only difference between the two is that a function header is not followed by a semicolon. The list of variable in the function header is known as the formal parameter list. The parameter list may have zero or more parameters of any datatype. The function body contains instructions to perform the desired computation in a function.

The function definition itself can act as an implicit function declaration. So the programmer may skip the function declaration statement in case the function is defined before being used.

Example:

```
/* function returning the min between two numbers */  
int min(int numb1, int numb2) {  
    /* local variable declaration */  
    int result;  
    if (numb1 < numb2)  
        result = numb1  
    else  
        result = numb2;  
    return result;  
}
```

Function Call

Function Call

The function call statement invokes the function. When a function is invoked the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

Function call statement has the following syntax:

```
Function_name(variable1, variable2,...);
```

When the function declaration is present before the function call, the compiler can check if the correct number and type of arguments are used in the function call and the returned value, if any, is being used reasonably.

Function definitions are often placed in a separate header file which can be included in other C source files that wish to use the functions. For example, the header file `stdio.h`, contains the definition of `scanf` and `printf` functions. We simply include this header file and call these functions without worrying about the code to implement their functionality.

List of Variable used in function call is known as actual parameter list. The actual parameter list may contain variable names, expressions, or constants.

Important points while calling a function:

1. Function name and the number and type of arguments in the function call must be same as that given in the function declaration and function header of the function definition.
2. If by mistake the parameters passed to a function are more than what it is specified to accept then the extra arguments will be discarded.
3. If by mistake the parameters passed to a function are less than what it is specified to accept then the unmatched argument will be initialized to some garbage value.
4. Names of variables in function declaration, function call, and header of function definition may vary.
5. If the data type of the argument passed does not match with that specified in the function declaration then either the unmatched argument will be initialized to some garbage value or compile time error will be generated.
6. Argument may be passed in the form of expressions to the called function. In such cases, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.
7. The parameter list must be separated with commas.
8. If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as shown below

```
variable_name = function_name(variable1, variable2,...);
```

Example:

Write a program to find product of two integer using functions.

```
#include<stdio.h>

//function declaration
int mul (int a, int b);

int main()
{
    int n1,n2,product=1;
    printf("Enter the first number");
    scanf("%d",&n1);
    printf("Enter the second number");
    scanf("%d",&n2);
    product = mul(n1,n2);
    //function call
    printf("\n Product = %d",product);
    return 0;
}

//function definition
int sum(int a, int b)          // function header
{                              // function body
    int result;
    result = a+b;
    return result;
}
```

Output:

Enter the first number : 30

Enter the second number : 30

product = 90

The variable declared within the function and its parameters are local to that function. The programmer may use same names for variable in other functions. This eliminates the need for thinking and keeping unique names for variable declared in all the function in the program.

In the function `mul()`, we have declared variable `result` just like any other variable. Variable declared within a function are called automatic local variables because of two reasons.

- First, they are local to the function. So, their effect is limited to the function. Any change made to these variables is visible only in that function.
- Second, they are automatically created whenever the function is called and they cease to exist at the end of the function.

Parameter Passing

Function arguments

Basically, there are two types of arguments:

Actual arguments

Formal arguments

```
#include <stdio.h>
return_type tunc_name(arguments);
{
    .....
    .....
}

Int main()
{
    .....
    tunc_name(arguments_value);
    .....
    return 0;
}
```

The diagram illustrates the flow of arguments between a function definition and its call. In the function definition, the parameter is labeled 'arguments'. In the function call, the argument is labeled 'arguments_value'. An upward arrow labeled 'formal arguments' points from the 'arguments' parameter in the function call to the 'arguments' parameter in the function definition. A downward arrow labeled 'actual arguments' points from the 'arguments_value' argument in the function call to the 'arguments' parameter in the function definition.

Categories of functions

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories.

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

Function with no argument and no return value

When a function has no argument, it does not receive any data from the calling function, similarly, when it does not return a value, the calling function does not receive any data from the calling function. In effect, there is no data transfer between the calling function and the called function

Example:

```
#include <stdio.h>

void value(void);

void main()
{
    value();
}

void value(void)
{
    int year = 1, period = 5, amount = 5000, inrate = 0.12;
    float sum;
    sum = amount;
    while (year <= period) {
        sum = sum * (1 + inrate);
        year = year + 1;
    }
    printf(" The total amount is %f:", sum);
}
```

Function with argument and no return value

In category 1, the main function has no control over the way of function receive input data, we could make the calling function to read data from the terminal and pass it on the called function. This approach seems to be wiser because the calling function can check for the validity of data, if necessary, before it is handed over to the called function.

Example:

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

int main()
{
    int a = 20;
    int ar[5] = { 10, 20, 30, 40, 50 };
    char str[30] = "geeksforgeeks";
    function(a, &ar[0], &str[0]);
    return 0;
}
```

```
void function(int a, int* ar, char* str)
{
    int i;
    printf("value of a is %d\n\n", a);
    for (i = 0; i < 5; i++) {
        printf("value of ar[%d] is %d\n", i, ar[i]);
    }
    printf("\nvalue of str is %s\n", str);
}
```

Function with no argument and return value

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

Example:

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

int sum();

int main()
{
    int num;
    num = sum();
    printf("\nSum of two given values = %d", num);
    return 0;
}

int sum()
{
    int a = 50, b = 80, sum;
    sum = sqrt(a) + sqrt(b);
    return sum;
}
```

Function with argument and return value

In this category, it receive data from the calling function through arguments and send back value.

Example:

```
#include <stdio.h>
#include <string.h>
int function(int, int[]);
int main()
{
    int i, a = 20;
    int arr[5] = { 10, 20, 30, 40, 50 };
    a = function(a, &arr[0]);
    printf("value of a is %d\n", a);
    for (i = 0; i < 5; i++) {
        printf("value of arr[%d] is %d\n", i, arr[i]);
    }
    return 0;
}
int function(int a, int* arr)
{
    int i;
    a = a + 20;
    arr[0] = arr[0] + 50;
    arr[1] = arr[1] + 50;
    arr[2] = arr[2] + 50;
    arr[3] = arr[3] + 50;
    arr[4] = arr[4] + 50;
    return a;
}
```

Passing Parameters to the function

When a function is called, the calling, function may have to pass some values to the called function. We have been doing this in the programming examples given so far. We will now learn the technicalities involved in passing arguments/parameters to the called function.

Basically, there are two types of arguments. They are

- **Actual arguments**
- **Formal arguments**

The variables declared in the function prototype or definition are known as **Formal arguments** and the values that are passed to the called function from the main function are known as **Actual arguments**.

Basically, arguments or parameters can be passed to the called function. They include:

- Call by value in which values of the variables are passed by the calling function to the called function. The programs that we have written so far all call the function using call by value method of passing parameters.
- Call by reference in which address of the variables are passed by the calling function to the called function.

Pass by Value:

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
/* function definition to swap the values */
void swap(int x, int y) {

    int temp;

    temp = x; /* save the value of x */
    x = y;    /* put y into x */
    y = temp; /* put temp into y */

    return;
}
```

Example:

call the function swap() by passing actual values

```
#include<stdio.h>

/* function declaration */

void swap(int x, int y);

int main () {

    /* local variable definition */

    int a = 100;

    int b = 200;

    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );

    /* calling a function to swap the values */

    swap(a, b);

    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );

    return 0;

}
```

Output:

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :100

After swap, value of b :200

Pass by Reference:

The **Pass by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
```

```
void swap(int *x, int *y) {
```

```
    int temp;
```

```
    temp = *x;    /* save the value at address x */
```

```
    *x = *y;      /* put y into x */
```

```
    *y = temp;    /* put temp into y */
```

```
    return;
```

```
}
```

Example:

call the function swap() by passing values by reference

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

```
/* function declaration */
```

```
void swap(int *x, int *y);
```

```
int main () {
```

```
    /* local variable definition */
```

```
    int a = 100;
```

```
    int b = 200;
```

```
    printf("Before swap, value of a : %d\n", a );
```

```
    printf("Before swap, value of b : %d\n", b );
```

```

/* calling a function to swap the values.
   * &a indicates pointer to a ie. address of variable a and
   * &b indicates pointer to b ie. address of variable b.
   */
swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}

```

Output:

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100

Advantages of passing by reference:

- References allow a function to change the value of the argument, which is sometimes useful. Otherwise, constant references can be used to guarantee the function won't change the argument.
- Because a copy of the argument is not made, pass by reference is fast, even when used with large structs or classes.
- References can be used to return multiple values from a function (via out parameters).
- References must be initialized, so there's no worry about null values.

Disadvantages of passing by reference:

- Because a non-constant reference cannot be initialized with a constant l-value or an reference value (e.g. a literal or an expression), arguments to non-constant reference parameters must be normal variables.
- It can be hard to tell whether an argument passed by non-constant reference is meant to be input, output, or both. Judicious use of constant and a naming suffix for out variables can help.
- It's impossible to tell from the function call whether the argument may change. An argument passed by value and passed by reference looks the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.

Built-in Functions (String Functions)

Built-In function (String Function)

string.h header file supports all the string functions in C language. All the string functions are given below.

String functions	Description
<u>strcat ()</u>	Concatenates str2 at the end of str1
<u>strncat ()</u>	Appends a portion of string to another
<u>strcpy ()</u>	Copies str2 into str1
<u>strncpy ()</u>	Copies given number of characters of one string to another
<u>strlen ()</u>	Gives the length of str1
<u>strcmp ()</u>	Returns 0 if str1 is same as str2. Returns <0 if str1 < str2. Returns >0 if str1 > str2
<u>strcmpi ()</u>	Same as strcmp() function. But, this function negotiates case. "A" and "a" are treated as same.
<u>strchr ()</u>	Returns pointer to first occurrence of char in str1
<u>strrchr ()</u>	last occurrence of given character in a string is found
<u>strstr ()</u>	Returns pointer to first occurrence of str2 in str1
<u>strrstr ()</u>	Returns pointer to last occurrence of str2 in str1
<u>strdup ()</u>	Duplicates the string
<u>strlwr ()</u>	Converts string to lowercase
<u>strupr ()</u>	Converts string to uppercase
<u>strrev ()</u>	Reverses the given string
<u>strset ()</u>	Sets all character in a string to given character
<u>strnset ()</u>	It sets the portion of characters in a string to given character
<u>strtok ()</u>	Tokenizing given string using delimiter

strcat():

strcat() function in C language concatenates two given strings. It concatenates source string at the end of destination string.

Syntax for strcat() function is given below.

```
char * strcat ( char * destination, const char * source );
```

Example:

strcat (str2, str1); – str1 is concatenated at the end of str2.

strcat (str1, str2); – str2 is concatenated at the end of str1.

- As you know, each string in C is ended up with null character ('\0').

- In strcat() operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strcat() operation.

Example Program:

```
#include <stdio.h>

#include <string.h>

int main( )
{
    char source[ ] = "Welcome to " ;
    char target[ ]= " C Programming" ;
    printf ( "\nSource string = %s", source ) ;
    printf ( "\nTarget string = %s", target ) ;
    strcat ( target, source ) ;
    printf ( "\nTarget string after strcat( ) = %s", target ) ;
}
```

Output:

```
Source string           = Welcome to
Target string           = C Programming
Target string after strcat( ) = C Programming Welcome to
```

Strncat()

strncat() function in C language concatenates (appends) portion of one string at the end of another string.

Syntax for strncat() function is given below.

```
char * strncat ( char * destination, const char * source, size_t num );
```

Example :

strncat (str2, str1, 3); – First 3 characters of str1 is concatenated at the end of str2.

strncat (str1, str2, 3); – First 3 characters of str2 is concatenated at the end of str1.

•As you know, each string in C is ended up with null character ('\0').

•In strncat() operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strncat() operation.

Example:

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

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

```
int main( )
```

```
{
```

```
    char source[ ] = " fresh2refresh" ;
```

```
    char target[ ]= "C tutorial" ;
```

```
    printf ( "\nSource string = %s", source ) ;
```

```
    printf ( "\nTarget string = %s", target ) ;
```

```
    strncat ( target, source, 5 ) ;
```

```
    printf ( "\nTarget string after strncat( ) = %s", target ) ;
```

```
}
```

Output:

Source string = Welcome

Target string = C Programming

Target string after strncat() = C Programming Welc

Strcpy()

strcpy() function copies contents of one string into another string.

Syntax for strcpy function is given below.

```
char * strcpy ( char * destination, const char * source );
```

Example:

strcpy (str1, str2) – It copies contents of str2 into str1.

strcpy (str2, str1) – It copies contents of str1 into str2.

•If destination string length is less than source string, entire source string value won't be copied into destination string.

For example, consider destination string length is 10 and source string length is 20. Then, only 10 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

Example:

```
#include <stdio.h>

#include <string.h>

int main( )
{
    char source[ ] = "Welcome" ;
    char target[20]= "" ;
    printf ( "\nsource string = %s", source ) ;
    printf ( "\ntarget string = %s", target ) ;
    strcpy ( target, source ) ;
    printf ( "\ntarget string after strcpy( ) = %s", target ) ;
    return 0;
}
```

Output:

```
Sourcestring = Welcome
target string =
target string after strcpy( ) = Welcome
```

Strlen():

strlen() function in C gives the length of the given string.
Syntax for strlen() function is given below.

```
size_t strlen ( const char * str );
```

- strlen() function counts the number of characters in a given string and returns the integer value.
- It stops counting the character when null character is found. Because, null character indicates the end of the string in C.

Example:

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

int main( )
{
    int len;
    char array[20]="Programming" ;
    len = strlen(array) ;
    printf ( "\string length = %d \n" , len ) ;
    return 0;
}
```

Output:

String Length = 11

Strcmp():

strcmp() function in C compares two given strings and returns zero if they are same.

If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value.

Syntax for strcmp() function is given below.

```
int strcmp ( const char * str1, const char * str2 );
```

strcmp() function is case sensitive. i.e, "A" and "a" are treated as different characters.

Example:

In this program, strings "fresh" and "refresh" are compared. 0 is returned when strings are equal. Negative value is returned when str1 < str2 and positive value is returned when str1 > str2.


```
#include <stdio.h>
#include <string.h>
int main( )
{
    char str1[ ] = "fresh" ;
    char str2[ ] = "refresh" ;
    int i, j, k ;
    i = strcmp ( str1, "fresh" ) ;
    j = strcmp ( str1, str2 ) ;
    k = strcmp ( str1, "f" ) ;
    printf ( "\n%d %d %d", i, j, k ) ;
    return 0;
}
```

Output:

0-11

strchr()

strchr() function returns pointer to the first occurrence of the character in a given string.

Syntax for strchr() function is given below.

```
char *strchr(const char *str, int character);
```

Example:

In this program, strchr() function is used to locate first occurrence of the character 'i' in the string "This is a string for testing". Character 'i' is located at position 3 and pointer is returned at first occurrence of the character 'i'.

```

#include <stdio.h>
#include <string.h>

int main ()
{
    char string[55] ="This is a string for testing";
    char *p;
    p = strchr (string,'i');
    printf ("Character i is found at position %d\n",p-string+1);
    printf ("First occurrence of character \'i\' in \'%s\' is \' \'%s\'",string, p);
    return 0;
}

```

Output:

Character i is found at position 3
 First occurrence of character "i" in "This is a string for testing" is "is is a string for testing"

Strdup():

strdup() function in C duplicates the given string.

Syntax for strdup() function is given below.

```
char *strdup(const char *string);
```

strdup() function is non standard function which may not available in standard library in C.

Example:

In this program, string "Welcome" is duplicated using strdup() function and duplicated string is displayed as

```

#include <stdio.h>
#include <string.h>
int main()
{
    char *p1 = "Welcome";
    char *p2;
    p2 = strdup(p1);
    printf("Duplicated string is : %s", p2);
    return 0;
}

```

Output:

Duplicated string is : Welcome

Strlwr():

strlwr() function converts a given string into lowercase.

Syntax for strlwr() function is given below.

```
char *strlwr(char *string);
```

strlwr() function is non standard function which may not available in stand library in C.

Example:

In this program, string " PROGRAMMING In C " is converted into lower case using strlwr() function and result is displayed as "programming in c".

```
#include<stdio.h>

#include<string.h>

int main()
{
    char str[ ] = "PROGRAMMING In C";
    printf("%s\n",strlwr (str));
    return 0;
}
```

Output:

programming in c

Strupr():

strupr() function converts a given string into uppercase.

Syntax for strupr() function is given below.

```
char *strupr(char *string);
```

strupr() function is non standard function which may not available in stand library in C.

Example:

In this program, string "programming IN c" is converted into uppercase using strupr() function and result is displayed as "PROGRAMMING IN C".

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[ ] = "programming In C";
    printf("%s\n",strupr(str));
    return 0;
}
```

Output:

PROGRAMMING IN C

Strrev():

strrev() function reverses a given string in C language.

Syntax for strrev() function is given below

```
char *strrev(char *string);
```

strrev() function is non standard function which may not available in standard library in C.

Example:

In below program, string "Hello" is reversed using strrev() function and output is displayed as "olleH".

```
#include<stdio.h>
#include<string.h>
void main()
{
    char name[30] = "Hello";
    printf("String before strrev( ) : %s\n",name);
    printf("String after strrev( ) : %s",strrev(name));
}
```

Output:

String before strrev() : Hello

String after strrev() : olleH

Strncpy():

strncpy() function copies portion of contents of one string into another string. Syntax for strncpy() function is given below.

```
char * strncpy ( char * destination, const char * source, size_t num );
```

Example:

strncpy (str1, str2, 4) – It copies first 4 characters of str2 into str1.

strncpy (str2, str1, 4) – It copies first 4 characters of str1 into str2.

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

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

```
int main( )
```

```
{
```

```
    char source[ ] = "welcome2c" ;
```

```
    char target[20]= "" ;
```

```
    printf ( "\nsource string = %s", source ) ;
```

```
    printf ( "\ntarget string = %s", target ) ;
```

```
    strncpy ( target, source, 6 ) ;
```

```
    printf ( "\ntarget string after strcpy( ) = %s", target ) ;
```

```
    return 0;
```

```
}
```

Output:

```
source string = welcome2c
```

```
target string =
```

```
target string after strncpy( ) = welcom
```

Strcmpi()

strcmpi() function in C is same as strcmp() function. But, strcmpi() function is not case sensitive. i.e, "A" and "a" are treated as same characters. Where as, strcmp() function treats "A" and "a" as different characters.

strcmpi() function is non standard function which may not available in standard library in C.

Both functions compare two given strings and returns zero if they are same.

Example:

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char str1[ ] = "welcome" ;
    char str2[ ] = "rewelcome" ;
    int i, j, k ;
    i = strcmpi ( str1, "WELCOME" ) ;
    j = strcmpi ( str1, str2 ) ;
    k = strcmpi ( str1, "f" ) ;
    printf ( "\n%d %d %d", i, j, k ) ;
    return 0;
}
```

Output:

0-11

Strtok():

strtok() function in C tokenizes/parses the given string using delimiter.

Syntax for strtok() function is given below.

```
char * strtok ( char * str, const char * delimiters );
```

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char string[50] = "Test,string1,Test,string2:Test:string3";
    char *p;
    printf ("String \"%s\" is split into tokens:\n",string);
    p = strtok (string, ",:");
    while (p!= NULL)
    {
        printf ("%s\n",p);
        p = strtok (NULL, ",:");
    }
    return 0;
}
```

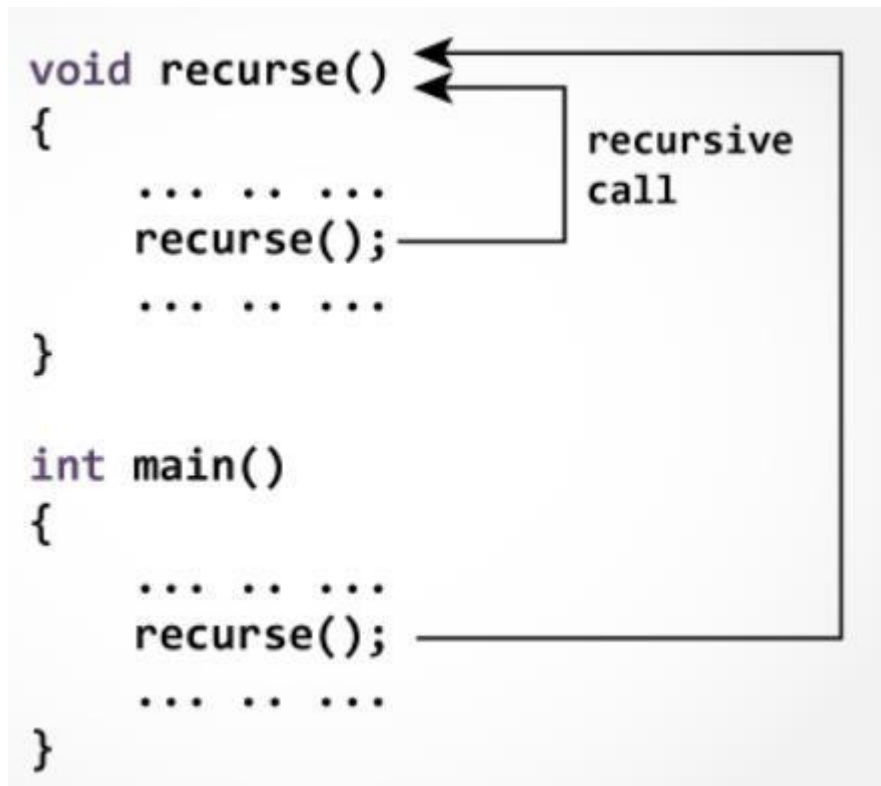
Output:

```
String "Test,string1,Test,string2:Test:string3" is split into tokens:
Test
string1
Test
string2
Test
string3
```

Recursive Functions

Recursive Functions:

A function that calls itself is known as a recursive function. And, this technique is known as recursion.



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

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Example:

Write a C program to find Factorial of a number using Recursive Function

```
#include <stdio.h>

unsigned long int factorial(unsigned int i) {
    if(i <= 1) {
        return 1;
    }
    return i * factorial(i - 1);
}

int main() {
    int i = 5;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

Output:

Factorial of 5 is 120

Write a C Program to find Fibonacci series for a given number using a recursive function

```
#include <stdio.h>

int fibo(int i) {
    if(i == 0) {
        return 0;
    }
    if(i == 1) {
        return 1;
    }
    return fibo(i-1) + fibo(i-2);
}
```

```
int main() {  
    int x;  
    for (x = 0; x < 9; x++) {  
        printf("%d\t\n", fibo(x));  
    }  
    return 0;  
}
```

Output:

```
0  
1  
1  
2  
3  
5  
8  
13  
21
```

Exercise Programs

Exercise Programs:

Calculate the total amount of power consumed by 'n' devices (Passing an array to a function):

```
#include <stdio.h>

#include <stdlib.h>

int calc_Electricity();//function prototype

int devices(int n[],int size);

int main()

{
    int size,i,n[50],s;

    printf("enter the size of an array");

    scanf("%d",&size);

    s=devices(n ,size);

    printf("the value is",s);

    return 0;

}

int calc_Electricity(int unit){//function definition

printf("Enter total units consumed\n");

scanf("%d",&unit);

double amount;

if((unit>=1)&&(unit<=50))//between 1 - 50 units

{

amount=unit*1.50;

}
```

```
else if((unit>50)&&(unit<=150))//between 50 150 units
{
amount=((50*1.5)+(unit-50)*2.00);
}
else if((unit>150)&&(unit<=250)){//between 150 - 250 units
amount=(50*1.5)+((150-50)*2.00)+(unit-150)*3.00;
}
else if(unit>250){//above 250 units
amount=(50*1.5)+((150-50)*2.00)+((250-150)*3.00)+(unit-250)*4;
}
else{
printf("No usage ");
amount=0;
}
//printf("Electricity bill = Rs. %.2f",amount);
return amount;
}

int devices(int n[],int size)
{
    int total=0;
    int i;
    int unit=0;
    int p;
    for (i=0;i<size;i++)
```

```
{  
p=calc_Electricity(unit);  
printf("the amount of one bill %d is",p);  
n[i]=p;  
total=total+n[i];  
printf("the total amount of n devices is %d",total);  
}  
}
```

Output:

Enter the number of devices : 3

Enter total unit consumed : 200

The total amount of 1 device is 425, Total amount of n device is 425

Enter total unit consumed : 250

The total amount of 1 device is 575, Total amount of n device is 1000

Enter total unit consumed : 200

The total amount of 1 device is 425, Total amount of n device is 1425

Exercise:

Menu-driven program to count the numbers which are divisible by 3, 5 and both (passing an array to a function)

```
#include<stdio.h>

int menudriven(int a[]);

int main()
{
    int k,s,i;

    int a[3]={0,1,2};

    menudriven(a);
}

int menudriven(int a[])
{
    int i,n,s,p,j;

    printf("Enter the value of n");

    scanf("%d",&n);

    for(j=0;j<3;j++){

        switch(a[j])

        {

        case 0:

            for (i=1; i<=n; i++)

                {

                    if(i%5==0)

                        {

                            p=p+1;}}

        printf("\nth total numbers divisible by 5 is %d",p);

        case 1:
```



```

p=0;
for (i=1; i<=n; i++)
{
    if (i%3==0)
    {
        p=p+1;}}
printf("\nthe total numbers divisible by 3 is %d",p);

```

case 2:

```

p=0;
for (i=1; i<=n; i++)
{
    if (i%3==0&& i%5==0)
    {
        p=p+1 ;}
}
printf("\nthe total numbers divisible by both are %d",p);
}
}}
```

Output:

Enter the value of n : 100

The total numbers divisible by 5 is : 20

The total numbers divisible by 3 is : 33

The total numbers divisible by both is : 6

Quiz

1. In C programming, parameters are always

- a. Passed by value
- b. Passed by reference
- c. Non-pointers are passed by value and pointers are passed by reference
- d. Passed by value result

2. Which of the following is true about return type of functions in C?

- a. Functions can return any type
- b. Functions can return any type except array and functions
- c. Functions can return any type except array, functions and union
- d. Functions can return any type except array, functions, function pointer and union

3. Predict Output.

```
#include <stdio.h>

int main()
{
    printf("%d", main);
    return 0;
}
```

- a. Address of main function
- b. Compiler Error
- c. Runtime Error
- d. Some random value

4. In C program, what is the meaning of following function prototype with empty parameter list.

```
void fun()
{
    /*...*/
}
```

- a. Function can only be called without any parameter
- b. Function can be called with any number of parameters of any types
- c. Function can be called with any number of integer paramete
- d. Function can be called with one integer parameter.

5. Output of the following

```
#include<stdio.h>

void dynamic(int s, ...)
{
    printf("%d ", s);
}

int main()
{
    dynamic(2, 4, 6, 8);
    dynamic(3, 6, 9);
    return 0;
}
```

- a. 2 3
- b. Compiler Error
- c. 4 3
- d. 3 2

6. Predict Output.

```
#include <stdio.h>
int main()
{
    int (*ptr)(int ) = fun;
    (*ptr)(3);
    return 0;
}

int fun(int n)
{
    for(;n > 0; n--)
        printf("C Program");
    return 0;
}
```

- a. C Program C Program C program
- b. C Program C program
- c. Compiler Error
- d. Runtime Error

7. Use of function

- a. Make the debugging task easier
- b. Helps to avoid repeating a set of statements many times
- c. Enhance the logical clarity of the program
- d. All the above

8. What is function

- a. A function is a block of statement that perform some specific task
- b. Function is a fundamental modular unit. A function is usually designed to perform a specific task
- c. Function is a block of code that perform a specific task. It has a name and it is reusable.
- d. All the above.

9. Default parameter passing mechanism is

- a. Call by value
- b. Call by reference
- c. Call by value result
- d. None of the above.

10. Which of the following is the complete function?

- a. `int funct();`
- b. `int funct(int x) { return x=x+1; }`
- c. `void funct(int) { printf("Hello"); }`
- d. Non of the above

11. The recursive function executed in a

- a. Parallel order
- b. First in first out order
- c. Last in first out order
- d. Iterative order

12. Which function will you choose to join two words?

- a) strcpy()
- b) strcat()
- c) strncon()
- d) memcon()

13. The _____ function appends not more than n characters.

- a) strcat()
- b) strcon()
- c) strncat()
- d) memcat()

14. What will strcmp() function do?

- a) compares the first n characters of the object
- b) compares the string
- c) undefined function
- d) copies the string

15. What is the prototype of strcoll() function?

- a) int strcoll(const char *s1,const char *s2)
- b) int strcoll(const char *s1)
- c) int strcoll(const *s1,const *s2)
- d) int strcoll(const *s1)

16. What is the function of strcoll()?

- a) compares the string, result is dependent on the LC_COLLATE
- b) copies the string, result is dependent on the LC_COLLATE
- c) compares the string, result is not dependent on the LC_COLLATE
- d) copies the string, result is not dependent on the LC_COLLATE

17. Which of the following is the variable type defined in header string. h?

- a) sizet
- b) size
- c) size_t
- d) size-t

18. What is the use of function `char *strchr(ch, c)`?

- a) return pointer to first occurrence of `ch` in `c` or `NULL` if not present
- b) return pointer to first occurrence of `c` in `ch` or `NULL` if not present
- c) return pointer to first occurrence of `ch` in `c` or ignores if not present
- d) return pointer to first occurrence of `c` in `ch` or ignores if not present

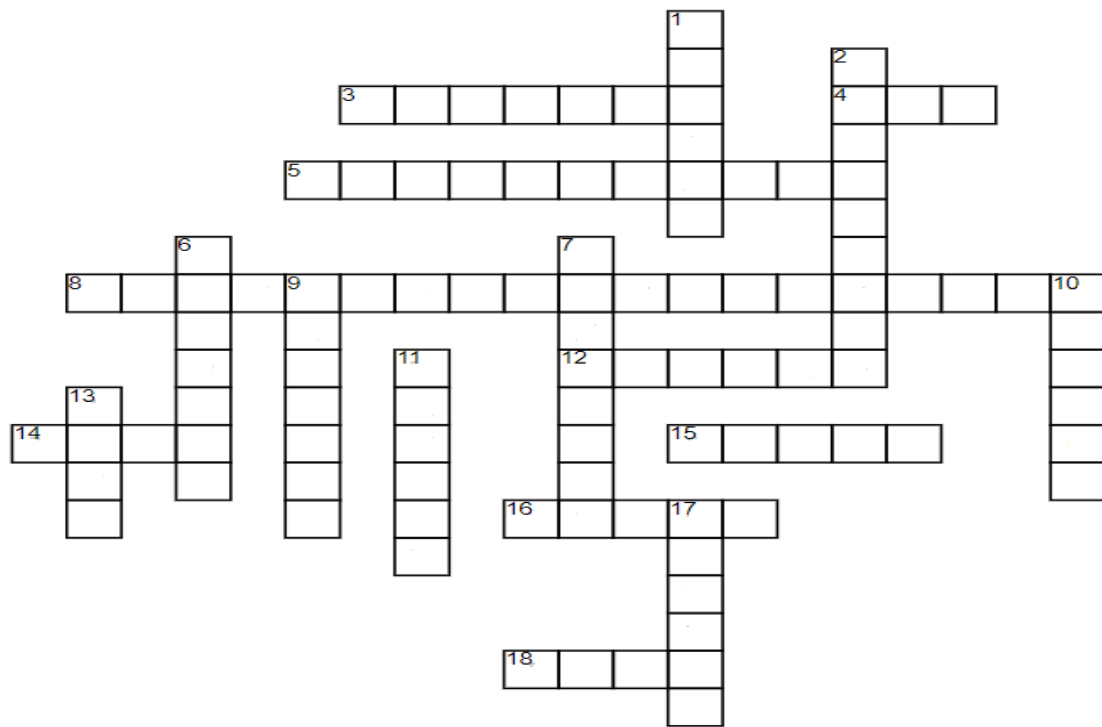
19. The `mem` functions are meant for _____

- a) returning a pointer to the token
- b) manipulating objects as character arrays
- c) returning a pointer for implemented-defined string
- d) returning a pointer to first occurrence of string in another string

20. What is the function of `void *memset(s, c, n)`?

- a) places character `s` into first `n` characters of `c`, return `c`
- b) places character `c` into first `n` characters of `s`, return `s`
- c) places character `s` into first `n` characters of `c`, return `s`
- d) places character `c` into first `n` character of `s`, return `c`

Crossword



Across

3. Which function returns true only for the characters defined as lowercase letters?
4. What is the default return type if it is not specified in function definition?
5. The default parameter passing mechanism is
8. The recursive functions are executed in a
12. What is the return-type of the function sqrt()?
14. NULL is the macro defined in the header string. h. (true or false)
15. Which of the following is the variable type defined in header string. h?
16. When a function is recursively called all the automatic variables are stored in a
18. Functions can return enumeration constants in C?

Down

1. Which function will you choose to join two words?
2. Functions have
6. Which function tests for any character that is an uppercase letter.
7. The_____function tests for any hexadecimal-digit character.
9. The_____function tests for any character for which isalpha or isdigit is true.
10. The value obtained in the function is given back to main by using _____ keyword.
11. Which among the following is Copying function?
13. Functions can return structure in C?
17. Which header declares several functions useful for testing and mapping characters?

Assignment

Unit IV

Assignment Questions

CO 1	Develop C program solutions to simple computational problems		
1.	<p>Write a program in C to find the Sum and Average of two numbers using function.</p> <p>Test Data :</p> <p>Enter the first number : 10</p> <p>Enter the second number : 20</p> <p>Expected Output :</p> <p>The sum of two number is : 30</p> <p>The Average of two number is : 15</p>	K2	CO4
2.	<p>Write a program in C to swap two numbers using function.</p> <p>Test Data :</p> <p>Enter the first number : 10</p> <p>Enter the second number : 20</p> <p>Expected Output :</p> <p>Before swapping : (10,20)</p> <p>After swapping : (20,10)</p>	K2	CO4

Part A

Question & Answer

Part A

1. What is Function? CO4 (K3)

A function definition in C programming consists of a function header and a function body. Return Type – A function may return a value.

2. List out the type of function in C programming. CO4 (K3)

Built-in Function

User-Defined Function

3. Give the syntax of function prototype. CO4 (K3)

returnType functionName(type1 argument1, type2 argument2, ...);

4. Give the syntax of calling function. CO4 (K3)

functionName(argument1, argument2, ...);

5. Write a syntax for Function with arguments and return value CO4 (K3)

Function declaration : int function (int);

Function call : function(x);

Function definition:

```
int function( int x )
```

```
{
```

```
statements;
```

```
return x;
```

```
}
```

6. What is static function? CO4 (K3)

They can directly refer to other static members of the class.

Static member functions do not have this pointer.

Static member function can not be virtual.

7. What is recursion? CO4 (K3)

Recursion is a common method of simplifying a problem into subproblems of same type. This is called divide and conquer technique. A basic **example** of **recursion** is factorial function.

8. Difference between strcmpi() and strncmp()? CO4 (K3)

strcmp compares both the strings till null-character of either string comes whereas strncmp compares at most num characters of both strings. But if num is equal to the length of either string than strncmp behaves similar to strcmp

9. Difference between the formal argument and the actual argument

CO4 (K3)

The major difference between actual and formal arguments is that actual arguments are the source of information; calling programs pass actual arguments to called functions. The called functions access the information using corresponding formal arguments. The following piece of code demonstrates actual and formal arguments.

10. Write a Syntax of return statement. CO4 (K3)

```
return (expression);
```

11. How arguments are passed to functions in C? CO4 (K3)

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

12. What is main() function? CO4 (K3)

In C, the "main" function is treated the same as every function, it has a return type (and in some cases accepts inputs via parameters). The only difference is that the main function is "called" by the operating system when the user runs the program.

13. What is pre-defined function? CO4 (K3)

predefined function (plural predefined functions) (computing) Any of a set of subroutines that perform standard mathematical functions included in a programming language; either included in a program at compilation time, or called when a program is executed.

Part B

Questions

Part B

1. What is a user defined function? Why it is used? CO4 (K3)
2. Define Recursion function? CO4 (K3)
3. List the Function Prototypes and explain it with examples CO4 (K3)
4. Tell in detail how an array can be passed as a parameter in user defined function. Give an example program CO4 (K3)
5. Recall the various types of functions supported by C. Give examples for each of the C functions. CO4 (K3)
6. Summarize the rules that apply to a function call in C. what relationship must be maintained between actual arguments and formal argument? CO4 (K3)
7. Outline predefined function and user defined function with example CO4 (K3)
8. Illustrate a program to find the factorial of a number using recursion CO4 (K3)
9. Identify the rules in regard to a function in C and Write a recursive function to evaluate the factorial of a number n CO4 (K3)
10. Build a function to reverse a given string and use it to check whether the given string is a palindrome. CO4 (K3)
11. Construct a program to sort the array of elements in ascending order using functions CO4 (K3)
12. Distinguish the following i)Global and local variables (6) ii)Automatic and static variables CO4 (K3)
13. Inspect a program to find the biggest of the given three values and use it to find the total obtained by a student which in turn is the sum of the best of three test scores and the best of three assignment scores CO4 (K3)
14. Write a c program to assess the reverse() function which accepts a string and display it in reverse CO4 (K3)
15. Create a C program to replace the punctuations from a given sentence by the space character using passing an array to a function CO4 (K3)

Supportive Online Certification

Unit IV

Certification Courses

⚙ NPTEL

Problem solving through Programming in C

<https://nptel.ac.in/courses/106/105/106105171/>

⚙ Coursera

1) C for Everyone: Structured Programming

<https://www.coursera.org/learn/c-structured-programming>

2) C for Everyone: Programming Fundamentals

<https://www.coursera.org/learn/c-for-everyone>

Real time Applications

Unit IV

Functions are used at many places in real life applications and some applications are listed here.

- ✿ Functions, commonly used in calculation or computation purpose
- ✿ To repeated use of calculation we can use functions
- ✿ To calculate area and circumferences of different shapes.
- ✿ Major imaging activity
- ✿ String based activity and simple calculation too we are using functions in the form of built-in or user defined function.
- ✿ Pointer related calculation

Content beyond syllabus

Unit IV

Content beyond syllabus

- 1) Comparison between with using string built-in function and without using string built-in function
- 2) Argument of Functions

Assessment Schedule

Unit IV

Prescribed Text book & References

Unit IV

Text books & References

TEXT BOOK

1. Reema Thareja, "Programming in C", Oxford University Press, Second Edition, 2016

REFERENCES:

1. Kernighan, B.W and Ritchie,D.M, "The C Programming language", Second Edition, Pearson Education, 2006
2. Paul Deitel and Harvey Deitel, "C How to Program", Seventh edition, Pearson Publication
3. Juneja, B. L and Anita Seth, "Programming in C", CENGAGE Learning India pvt. Ltd., 2011
4. Pradip Dey, Manas Ghosh, "Fundamentals of Computing and Programming in C", First Edition, Oxford University Press, 2009
5. ER and ETA , "CC Foundation Program Reference materials" Infosys Ltd.

Mini Project Suggestions

Unit IV

- 1) Scientific calculator
- 2) Library Management System

Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

OCS752

INTRODUCTION TO C

PROGRAMMING

Department: : Electrical and Electronics Engineering

Batch/Year: 2017-2021

Created by: Dr. S. Meenakshi and A.S. Vibith

Date: 21-10-2020

Table of Contents

- ✿ Course Objectives
- ✿ Syllabus
- ✿ Course Outcomes (Cos)
- ✿ CO-PO Mapping
- ✿ Lecture Plan
- ✿ Activity based learning
- ✿ Lecture notes
- ✿ Assignments
- ✿ Part A Q&A
- ✿ Part B Qs
- ✿ List of Supportive online Certification courses
- ✿ Real time applications in day to day life and to industry
- ✿ Contents beyond Syllabus
- ✿ Assessment Schedule (proposed and actual date)
- ✿ Prescribed Text Books & Reference Books
- ✿ Mini Project Suggestions

Course Objectives

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C

3 0 0 3

OBJECTIVES

- ✿ To develop C Programs using basic programming constructs
- ✿ To develop C programs using arrays and strings
- ✿ To develop applications in C using functions and structures

Syllabus

OCS752 INTRODUCTION TO C PROGRAMMING

L T P C 3 0 0 3

UNIT I INTRODUCTION

9

Structure of C program – Basics: Data Types – Constants – Variables - Keywords – Operators: Precedence and Associativity - Expressions - Input/output statements, Assignment statements – Decision-making statements - Switch statement - Looping statements – Pre-processor directives - Compilation process – Exercise Programs: Check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers

UNIT II ARRAYS

9

Introduction to Arrays – One dimensional arrays: Declaration – Initialization - Accessing elements – Operations: Traversal, Insertion, Deletion, Searching - Two dimensional arrays: Declaration – Initialization - Accessing elements – Operations: Read – Print – Sum – Transpose – Exercise Programs: Print the number of positive and negative values present in the array – Sort the numbers using bubble sort - Find whether the given is matrix is diagonal or not.

UNIT III STRINGS

9

Introduction to Strings - Reading and writing a string - String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse – Substring – Insertion – Indexing – Deletion – Replacement – Array of strings – Introduction to Pointers – Pointer operators – Pointer arithmetic - Exercise programs: To find the frequency of a character in a string - To find the number of vowels, consonants and white spaces in a given text - Sorting the names.

UNIT IV FUNCTIONS

9

Introduction to Functions – Types: User-defined and built-in functions - Function prototype – Function definition - Function call - Parameter passing: Pass by value - Pass by reference - Built-in functions (string functions) – Recursive functions – Exercise programs: Calculate the total amount of power consumed by 'n' devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)

UNIT V STRUCTURES

9

Introduction to structures – Declaration – Initialization – Accessing the members – Nested Structures – Array of Structures – Structures and functions – Passing an entire structure – Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

TOTAL:45 PERIODS

Course Outcomes

- ✿ CO 1 - Develop algorithmic solutions to simple computational problems using basic constructs K1
- ✿ CO 2 - Develop simple applications in C using Control Constructs K2
- ✿ CO 3 - Design and implement applications using arrays K2
- ✿ CO 4 – Represent data using string and string operations K3
- ✿ CO 5 - Decompose a C program into functions and pointers K3
- ✿ CO 6 - Represent and write program using structure and union K3

CO – PO Mapping

CO	PO	Mapping Level	Justification
CO	PO	Mapping Level	Justification
CO1	PO1	2	Identify the data type and operators to solve the problem
CO1	PO2	2	Design the expression in an efficient way
CO1	PO3	2	Recognize the need of basic c Tokens-variables-constants
CO1	PO5	2	Apply the concept of control statements for simple solving the problem
CO1	PO12	1	Formulate the iterative statements for problem solving
CO2	PO1	3	Develop a complete program s for preprocessor directives
CO2	PO2	3	Recognize the implementation of simple problem solving with above concepts
CO3	PO3	2	Apply simple mathematical concepts for writing 1D arrays and its operations
CO3	PO5	2	Identify and formulate for the given problem using 2D and its operations
CO3	PO12	1	Design way of problem solving in Multi Dimensional arrays
CO4	PO1	3	Recognize the need of implementation in string
CO4	PO2	3	Apply logic to solve simple problem statement using string operations
CO4	PO3	3	Apply the knowledge to find the possible code for string manipulations
CO5	PO5	2	Identify the code for decomposition as function
CO5	PO12	1	Develop functions and reuse it whenever required to reduce the lines of code
CO5	PO1	2	Recognize the need of function concepts
CO5	PO2	2	Apply compound data knowledge to select any one
CO5	PO3	2	Apply the concept of pointers
CO5	PO5	2	Design and Develop program using the selected compound data
CO6	PO12	1	Recognize the need of structure
CO6	PO1	2	Apply the basic idea of handling with union
CO6	PO2	2	Identify the number of modes and operations on structure in detail
CO6	PO3	2	Develop programs using structure and union

Lecture Plan

Unit V

CO-PO/PSO MAPPING

COURSE OUTCOME	LEVEL OF COURSE OUTCOME	PROGRAM OUTCOME (PO)												PROGRAM SPECIFIC OUTCOME (PSO)			
		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	K1	2	2	2	-	2	-	-	-	-	-	-	1	1			
CO2	K2	3	3	2	-	2	-	-	-	-	-	-	1	1			
CO3	K2	3	3	3	-	2	-	-	-	-	-	-	1	1			
CO4	K3	2	2	2	-	2	-	-	-	-	-	-	1	1			
CO5	K3	2	2	2	-	-	-	-	-	-	-	-	1	1			
CO6	K3	3	3	3		2							1	1			

Unit V - STRUCTURE

S.No	Topics	No. of Periods	Proposed Date	Actual Lecture Date	Pertaining CO	Taxonomy Level	Mode of Delivery
1	Introduction to structures – Declaration –	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
2	Initialization – Accessing the members –	1			CO1	K2	PPT, Chalk & Talk
3	Nested Structures –	1			CO1	K2	PPT, Chalk & Talk
4	Array of Structures –	1			CO1	K2	PPT, SHORT VIDEOS, Chalk & Talk
5.6	Structures and functions – Passing an entire structure –	2			CO1	K2	PPT, Chalk & Talk
7	Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) –	1			CO1	K2	PPT, Chalk & Talk
8,9	Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)	2			CO1	K2	PPT, Chalk & Talk

Activity Based Learning

Unit V

Activity Based Learning

- ✿ Learn by solving problems – Tutorial Sessions can be conducted
 - Tutorial sessions available in Skillrack for practice
- ✿ Learn by questioning
- ✿ Learn by doing hands-on IN ONLINR / VIRTUAL LAB.

Lecture Notes

UNIT V STRUCTURES

9

Introduction to structures – Declaration – Initialization – Accessing the members – Nested Structures – Array of Structures – Structures and functions – Passing an entire structure – Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

Unit V - Structure : LEARNING PLAN

Sl. No.	Topics	Learning Content (hh.mm)	Post-Session (Quiz + Assignment) (hh:mm)
5.0	Introduction to structures – Declaration – Initialization – Accessing the members –	3.00	1.00
5.1,5.2,5.3,5.4	Nested Structures – Array of Structures – Structures and functions – Passing an entire structure –	3.00	0.50
5.5	Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)	3.00	1.00
	Total	9.00	2.50

Introduction to structures – Declaration – Initialization – Accessing the members

Topic. 5.0

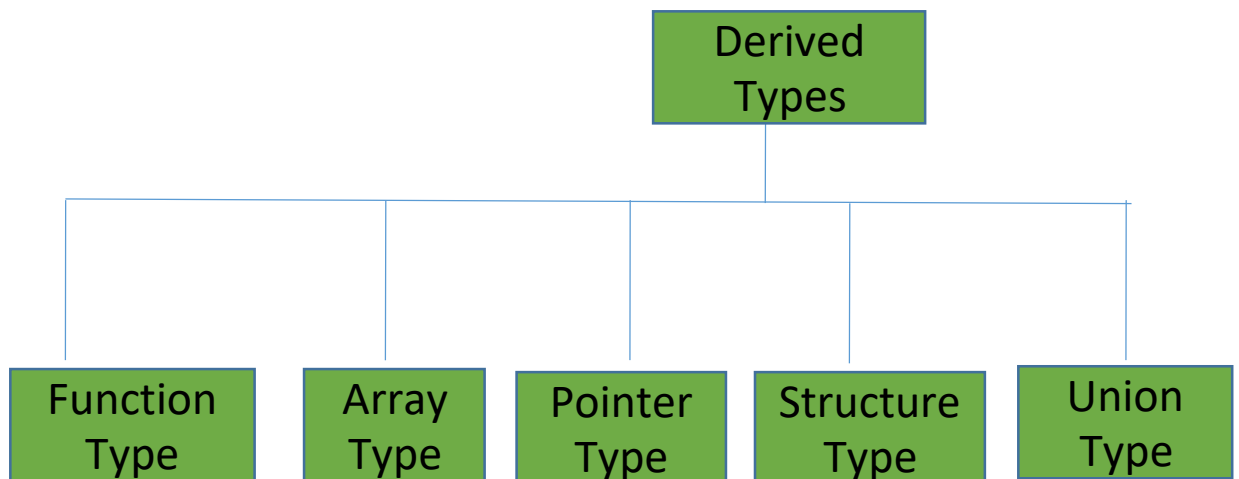
Introduction to structures–

C Data Types:

Primary data types

Derived data types

User-defined data types



Array – Collection of one or more related variables of similar data type grouped under a single name

Structure – Collection of one or more related variables of different data types, grouped under a single name

Need of structures

In a Library, each book is an **object**, and its **characteristics** like title, author, no of pages, price are grouped and represented by one **record**.

The characteristics are different types and grouped under a aggregate variable of different types.

A **record** is group of **fields** and each field represents one characteristic. In C, a record is implemented with a derived data type called **structure**. The characteristics of record are called the **members** of the structure.

Book-1

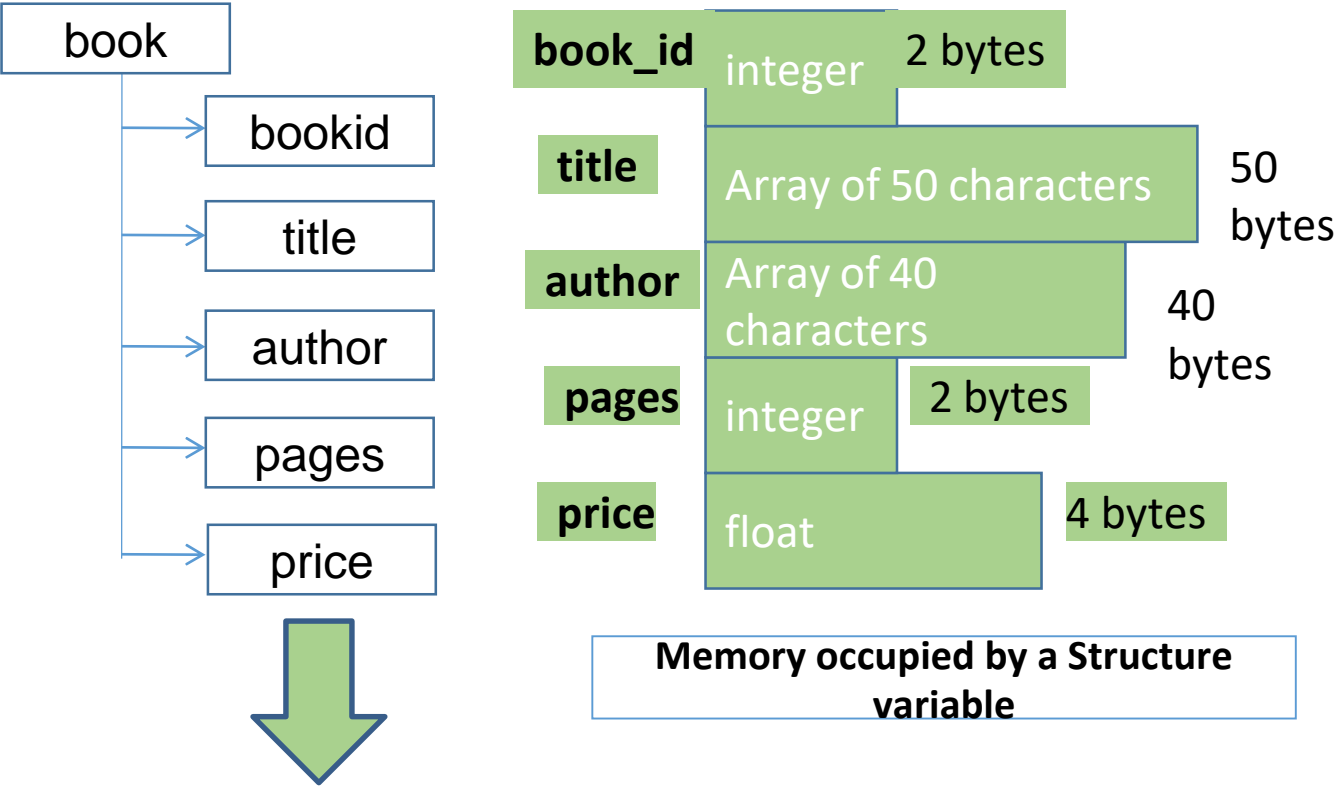
BookID: 1211
Title : C Primer Plus
Author : Stephen Prata
Pages : 984
Price : Rs. 585.00

Book-2

BookID: 1212
Title : The ANSI C Progg.
Author : Dennis Ritchie
Pages : 214
Price : Rs. 125.00

Book-3

BookID: 1213
Title : C By Example
Author : Greg Perry
Pages : 498
Price : Rs. 305.00



STRUCTURE- BOOK

struct book {

int book_id ;

char title[50] ;

char author[40] ;

int pages ;

float price ;

};

Structure tag

- A **Structure** is defined to be a collection of different data items, that are stored under a common name.
- A structure is same as that of records. It stores related information about an entity. Structure is basically a user defined data type that can store related information (even of different data types) together.

Declaration of structures

- A structure is declared using the keyword struct followed by a structure name. All the variables of the structures are declared within the structure. A structure type is defined by using the given syntax.

By declaring a structure type	By declaring a structure variable
struct struct-name {	struct stru-name Sv1,Sv2,Sv3;
data_type var-name;	(or)
data_type var-name;	struct stru-name {
...};	data_type var-name;
	data_type var-name;
	} sv1, sv2 , sv3;

Example :

```
struct student {
    int r_no;
    char name[20];
    char course[20];
    float fees; };
```

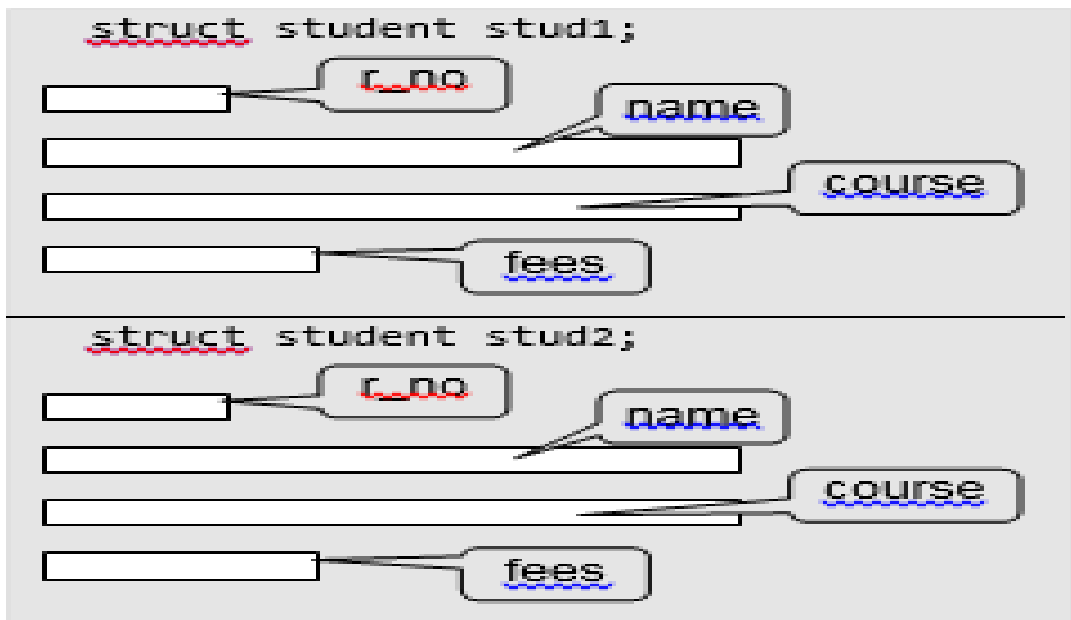
The structure definition does not allocates any memory. It just gives a template that conveys to the C compiler how the structure is laid out in memory and gives details of the member names. Memory is allocated for the structure when we declare a variable of the structure. For ex., we can define a variable of student by writing as :

```
struct student stud1;
```


Here, struct student is a data type and stud1 is a variable. Look at another way of declaring variables. In the following syntax, the variables are declared at the time of structure declaration.

```
struct student{  
int r_no;  
char name[20]; char course[20]; float fees;  
} stud1, stud2;
```

In this declaration we declare two variables stud1 and stud2 of the structure student. So if you want to declare more than one variable of the structure, then separate the variables using a comma. When we declare variables of the structure, separate memory is allocated for each variable. This is shown in Fig.



last but not the least, structure member names and names of the structure follow the same rules as laid down for the names of ordinary variables. However, care should be taken to ensure that the name of structure and the name of a structure member should not be the same. Moreover, structure name and its variable name should also be different.

Note: Structure type and variable declaration of a structure can be either local or global depending on their placement in the code.

Type def declarations

The typedef (derived from type definition) keyword enables the programmer to create a new data type name by using an existing data type. By using typedef, no new data is created, rather an alternate name is given to a known data type. The general syntax of using the typedef keyword is given as:

```
typedef existing_data_type new_data_type;
```

Note that typedef statement does not occupy any memory; it simply defines a new type. For example, if we write

```
typedef int INTEGER;
```

then INTEGER is the new name of data type int. To declare variables using the new data type name, precede the variable name with the data

type name (new). Therefore, to define an integer variable, we may now write

```
INTEGER num=5;
```

When we precede a struct name with typedef keyword, then the struct becomes a new type. It is used to make the construct shorter with more meaningful names for types already defined by C or for types that you have declared. With a typedef declaration, becomes a synonym for the type.

For example, writing

```
typedef struct student{
```

```
    int r_no;
```

```
    char name[20];
```

```
    char course[20];
```

```
    float fees;};
```

Now that you have preceded the structure's name with the keyword typedef, the student becomes a new data type. Therefore, now you can straight away declare variables of this new data type as you declare variables of type int, float, char, double, etc. to declare a variable of structure student you will just write,

```
student stud1;
```

Note that we have not written struct student stud1.

NOTE: Do not forget to place a semicolon after the declaration of structures and unions.

Accessing the members of a structure

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using a '.' (dot operator).

The syntax of accessing a structure a member of a structure is:

`struct_var.member_name`

`stud1.r_no`

membership operator



The dot operator is used to select a particular member of the structure. For example, to assign values to the individual data members of the structure variable `stud1`, we may write

```
stud1.r_no = 01;
```

```
stud1.name = "Rahul";
```

```
stud1.course = "BCA";
```

```
stud1.fees = 45000;
```

To input values for data members of the structure variable `stud1`, we may write

```
scanf("%d", &stud1.r_no);
```

```
scanf("%s", stud1.name);
```

Similarly, to print the values of structure variable `stud1`, we may write

```
printf("%s", stud1.course);
```

```
printf("%f", stud1.fees);
```

Memory is allocated only when we declare the variables of the structure. In other words, the memory is allocated only when we instantiate the structure. In the absence of any variable, structure definition is just a template that will be used to reserve memory when a variable of type `struct` is declared.

Once the variables of a structure are defined, we can perform a few operations on them. For example, we can use the assignment operator (`=`) to assign the values of one variable to another.

NOTE: Of all the operators `→`, `.`, `()`, and `[]` have the highest priority. This is evident from the following statement

`stud1.fees++` will be interpreted as `(stud1.fees)++`.

Initialization of structures

- Initializing a structure means assigning some constants to the members of the structure.
- When the user does not explicitly initialize the structure then C automatically does that. For int and float members, the values are initialized to zero and char and string members are initialized to the '\0' by default.
- The initializers are enclosed in braces and are separated by commas. Note that initializers match their corresponding types in the structure definition.
- The general syntax to initialize a structure variable is given as follows.

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .....
}struct_var = {constant1, constant2, constant 3,...};
OR
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .....
};
struct struct_name struct_var = {constant1, constant2, ....};
```

For example, we can initialize a student structure by writing,

```
struct student
{int r_no;
char name[20]; char course[20]; float fees;
}stud1 = {01, "Rahul", "BCA", 45000};
```

Or, by writing,

```
struct student stud1 = {01, "Rahul", "BCA", 45000};
```

Figure illustrates how the values will be assigned to individual fields of the structure.

<pre>struct student stud1 = {01, "Rahul", "BCA", 45000};</pre>				<pre>struct student stud2 = {07, "Rajiv"};</pre>			
01	Rahul	BCA	45000	07	Rajiv	\0	0.0
r_no	name	course	fees	r_no	name	course	fees

Assigning values to structure elements

When all the members of a structure are not initialized, it is called partial initialization. In case of partial initialization, first few members of the structure are initialized and those that are uninitialized are assigned default values

To Initialize or assign of structure variable **while declaration**

```
struct student stud1= {01, "Rahul", "BCA", 45000} ;
```

To initialize or assign value **to the individual data members** of the structure variable Rahul, we may write,

```
stud1.r_no = 01;  
strcpy(stud1.name, "Rahul");  
stud1.course = "BCA";  
stud1.fees = 45000;
```

Reading **values to members at runtime:**

```
struct student stud3;  
printf("\nEnter the roll no");  
scanf("%d",&stud3.r_no);  
printf("\nEnter the name");  
scanf("%s", stud3.name);  
printf("\nEnter the course");  
scanf("%s", stud3.course);  
printf("\nEnter the fees");  
scanf("%d",&stud3.fees);
```

We can initialize / assign a structure to another structure of the same type. For ex, if we have two structure variables stu1 and stud2 of type struct student given as

```
struct student stud1 = {01, "Rahul", "BCA", 45000};  
struct student stud2;
```

Then to assign one structure variable to another we will write,
stud2 = stud1;

Example Program 1: Write a program using structures to read and display the information about a student

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

struct employee {
    int empid;
    char name[35];
    int age;
    float salary;};

int main() {
    struct employee emp1 ;
    printf("Enter the details of employee 1 : ");
    scanf("%d %s %d %f" , &emp1.empid, emp1.name, &emp1.age, &emp1.salary);
    printf("Emp ID:%d\nName:%s\n Age:%d\n Salary:%f",emp1.empid,
    emp1.name, emp1.age,emp1.salary);}
```

Output :

```
Enter the details of employee 2 1212 Roit 29 20000
Employee1 is junior than Employee2
Emp ID:1211
Name:K.Ravi
Age:27
Salary:30000.000000

...Program finished with exit code 0
Press ENTER to exit console.
```

Example program 2: Write a program using structures to read student 3 marks and display the total and average of the student.

```
#include<stdio.h>
#include<conio.h>
struct stud
{
    int regno;
    char name[10];
    int m1;
    int m2;
    int m3;
};
struct stud s;
void main() {
    float tot,avg;
    printf("\nEnter the student regno,name,m1,m2,m3:");
    scanf("%d%s%d%d%d",&s.regno,&s.name,&s.m1,&s.m2,&s.m3);
    tot=s.m1+s.m2+s.m3;
    avg=tot/3;
    printf("\nThe student Details are:");
    printf("\n%d\t%s\t%f\t%f",s.regno,s.name,tot,avg);
}
```

Output :

Enter the student regno,name,m1,m2,m3:100

aaa

87

98

78

The student Details are:

100 aaa 263.000000 87.666664

GUIDED ACTIVITY – Here is the guided activity for you on (Implementing a Structure – declaration, initialization, accessing for an employee DB)

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

```
struct employee {
    int empid;
    char name[35];
    int age;
    float salary;
```

```
};
```

```
int main() {
    struct employee emp1, emp2 ;
```

```
    struct employee emp3 = { 1213 , "S.Murali" , 31 , 32000.00 } ;
```

```
    emp1.empid=1211;
```

```
    strcpy(emp1.name, "K.Ravi");
```

```
    emp1.age = 27;
```

```
    emp1.salary=30000.00;
```

```
    printf("Enter the details of employee 2");
```

```
    scanf("%d %s %d %f" , &emp2.empid, emp2.name,
    &emp2.salary);
```

```
    if(emp1.age > emp2.age)
```

```
        printf("Employee1 is senior than Employee2\n" );
```

```
    else
```

```
        printf("Employee1 is junior than Employee2\n");
```

```
    printf("Emp ID:%d\n Name:%s\nAge:%d\n Salary:%f",
    emp1.empid, emp1.name, emp1.age, emp1.salary);
```

```
}
```

Declaration of Structure Type

Declaration of Structure variables

Declaration and initialization of Structure variable

Initialization of Structure members individually

Reading values to members of Structure

Accessing members of Structure

Output:

```
Enter the details of employee 2 1212 Roit 29 20000
Employee1 is junior than Employee2
Emp ID:1211
Name:K.Ravi
Age:27
Salary:30000.000000

...Program finished with exit code 0
Press ENTER to exit console.
```


Copying and Comparing Structures

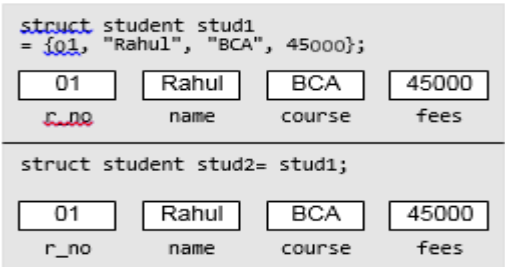
We can assign a structure to another structure of the same type. For example, if we have two structure variables stud1 and stud2 of type struct student given as

```
struct student stud1 = {01, "Rahul", "BCA", 45000};
```

```
struct student stud2;
```

Then to assign one structure variable to another, we will write

```
stud2 = stud1;
```



struct student stud1 = {01, "Rahul", "BCA", 45000};			
01	Rahul	BCA	45000
r_no	name	course	fees
struct student stud2= stud1;			
01	Rahul	BCA	45000
r_no	name	course	fees

Figure Values of structure variables

This statement initializes the members of stud2 with the values of members of stud1. Therefore, now the values of stud1 and stud2 can be given as shown in Fig.

C does not permit comparison of one structure variable with another. However, individual members of one structure can be compared with individual members of another structure. When we compare one structure member with another structure's member, the comparison will behave like any other ordinary variable comparison.

For example, to compare the fees of two students, we will write

```
if(stud1.fees > stud2.fees) //to check if fees of stud1 is greater than stud2
```

Note: An error will be generated if you try to compare two structure variables.

Nested Structures – Array of Structures – Structures and functions – Passing an entire structure –Passing Structures Through Pointers, Self referential structure

Topic. 5.1, 5.2, 5.3, 5.4

NESTED STRUCTURES

A structure can be placed within another structure. That is, a structure may contain another structure as its member. Such a structure that contains another structure as its member is called a nested structure.

Let us now see how we declare nested structures. Although it is possible to declare a nested structure with one declaration, it is not recommended. The easier and clearer way is to declare the structures separately and then group them in the higher level structure. When you do this, take care to check that nesting must be done from inside out (from lowest level to the most inclusive level), i.e., declare the innermost structure, then the next level structure, working towards the outer (most inclusive) structure.

```
typedef struct {
    char first_name[20];
    char mid_name[20];
    char last_name[20];
} NAME;
typedef struct {
    int dd;
    int mm;
    int yy;
} DATE;
typedef struct {
    int r_no;
    NAME name;
    char course[20];
    DATE DOB;
    float fees;
} student;
```

In this example, we see that the structure student contains two other structures, NAME and DATE. Both these structures have their own fields. The structure NAME has three fields: first_name, mid_name, and last_name. The structure DATE also has three fields: dd, mm, and yy, which specify the day, month, and year of the date. Now, to assign values to the structure fields, we will write

```
struct student stud1;
stud1.name.first_name = "Janak";
stud1.name.mid_name = "Raj";
stud1.name.last_name = "Thareja";
stud1.course = "BCA";
stud1.DOB.dd = 15;
stud1.DOB.mm = 09;
stud1.DOB.yy = 1990;
stud1.fees = 45000;
```

In case of nested structures, we use the dot operator in conjunction with the to access the members of the innermost as well as the outermost structures.

Guided activity on nested structures

```
#include<stdio.h>
#include<string.h>
struct date {
    int day ;
    int month ;
    int year ;
};
struct person {
    char name[40];
    int age ;
    struct date b_day ;
};
int main( ) {
    struct person p1;
    strcpy ( p1.name , "S. Ram" );
    p1.age = 32 ;
    p1.b_day.day = 25 ;
    p1.b_day.month = 8 ;
    p1.b_day.year = 1978 ;
}
```

Outer Structure

Inner Structure

Accessing Inner Structure members

OUTPUT:

No output since there is no print statement

Write a program to read and display information of a student using structure within a structure

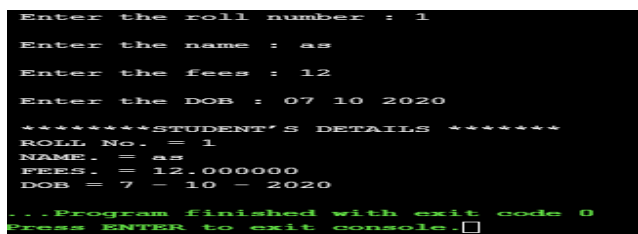
```
#include<stdio.h>

int main(){ struct DOB {
    int day;
    int month;
    int year;    };

    struct student {
        int roll_no;
        char name[100];
        float fees;
        struct DOB date;    };

    struct student stud1;
    printf("\n Enter the roll number : ");
    scanf("%d", &stud1.roll_no);
    printf("\n Enter the name : ");
    scanf("%s", stud1.name);
    printf("\n Enter the fees : ");
    scanf("%f", &stud1.fees);
    printf("\n Enter the DOB : ");
    scanf("%d %d %d", &stud1.date.day, &stud1.date.month, &stud1.date.year);
    printf("\n *****STUDENT'S DETAILS *****");
    printf("\n ROLL No. = %d", stud1.roll_no);
    printf("\n NAME. = %s", stud1.name);
    printf("\n FEES. = %f", stud1.fees);
    printf("\n DOB = %d - %d - %d", stud1.date.day, stud1.date.month,
stud1.date.year);
}
```

OUTPUT:



```
Enter the roll number : 1
Enter the name : as
Enter the fees : 12
Enter the DOB : 07 10 2020
*****STUDENT'S DETAILS *****
ROLL No. = 1
NAME. = as
FEES. = 12.000000
DOB = 7 - 10 - 2020
...Program finished with exit code 0
Press ENTER to exit console.□
```

Arrays Of Structures

In the above examples, we have seen how to declare a structure and assign values to its data members. Now, we will discuss how an array of structures is declared. For this purpose, let us first analyse where we would need an array of structures.

In a class, we do not have just one student. But there may be at least 30 students. So, the same definition of the structure can be used for all the 30 students. This would be possible when we make an array of structures. An array of structures is declared in the same way as we declare an array of a built-in data type.

Another example where an array of structures is desirable is in case of an organization. An organization has a number of employees. So, defining a separate structure for every employee is not a viable solution. So, here we can have a common structure definition for all the employees. This can again be done by declaring an array of structure employee.

The general syntax for declaring an array of structure can be given as,
`struct struct_name struct_var[index];`

Consider the given structure definition.

```
struct student{  
    int r_no;  
    char name[20]; char course[20]; float fees;};
```

A student array can be declared by writing,
`struct student stud[30];`

Now, to assign values to the i^{th} student of the class, we will write,
`stud[i].r_no = 09;
stud[i].name = "RASHI";
stud[i].course = "MCA";
stud[i].fees = 60000;`

In order to initialize the array of structure variables at the time of declaration, we can write as follows:

```
struct student stud[3] = {{01, "Aman", "BCA", 45000},{02, "Aryan", "BCA", 60000},  
{03,"John", "BCA", 45000}};
```

Write a program to read and display information of all the students in the class (using Array of structure)

```
#include<stdio.h>
int main()
{
    struct student
    {
        int roll_no;
        char name[80];
        float fees;
        char DOB[80];
    };
    struct student stud[50];
    int n, i;
    printf("\n Enter the number of students : ");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    {
        printf("Enter the roll number : ");
        scanf("%d", &stud[i].roll_no);
        printf("Enter the name : ");
        scanf("%s", stud[i].name);
        printf("Enter the fees : ");
        scanf("%f", stud[i].fees);
        printf("Enter the DOB : ");
        scanf("%s", stud[i].DOB);
    }
    for(i=0;i<n;i++)
    {
        printf("\n*DETAILS OF %dth STUDENT*", i+1);
        printf("\n ROLL No. = %d", stud[i].roll_no);
        printf("\n NAME. = %s", stud[i].name);
        printf("\n ROLL No. = %f", stud[i].fees);
        printf("\n ROLL No. = %s", stud[i].DOB);
    }
}
```

OUTPUT:

```
Enter the number of students : 2
Enter the roll number : 1
Enter the name : ashik
Enter the fees : 3500
Enter the DOB : 12-12-1978
Enter the roll number : 2
Enter the name : asmi
Enter the fees : 4500
Enter the DOB : 12-12-1990
```

```
*DETAILS OF 1th STUDENT*
ROLL No. = 1
NAME. = ashik
ROLL No. = 3500.000000
ROLL No. = 12-12-1978
*DETAILS OF 2th STUDENT*
ROLL No. = 2
NAME. = asmi
ROLL No. = 4500.000000
ROLL No. = 12-12-1990
```

...Program finished with exit code 0

GUIDED ACTIVITY – Here is the activity you on (arrays and structures)

```
struct student
{
    int sub[3] ;
    int total ;
};

int main( ) {
    struct student s[3];
    int i,j;
    for(i=0;i<3;i++) {
        printf("\n\nEnter student %d marks:",i+1);
        for(j=0;j<3;j++) {
            scanf("%d",&s[i].sub[j]);
        }
    }
    for(i=0;i<3;i++) {
        s[i].total =0;
        for(j=0;j<3;j++) {
            s[i].total +=s[i].sub[j];
        }
        printf("\nTotal marks of student %d is: %d",
            i+1,s[i].total );
    }
}
```

OUTPUT:

OUTPUT:

Enter student 1 marks: 60 60 60

Enter student 2 marks: 70 70 70

Enter student 3 marks: 90 90 90

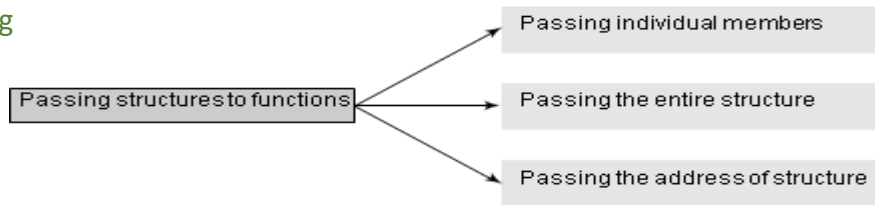
Total marks of student 1 is: 180

Total marks of student 2 is: 240

Total marks of student 3 is: 270

Structure and Functions

For structures to be fully useful, we must have a mechanism to pass them to functions and return them. A function may access the members of a structure in three ways as shown in Fig



Passing Individual Structure Members to a Function

To pass any individual member of the structure to a function we must use the direct selection operator to refer to the individual members for the actual parameters. The called program does not know if the two variables are ordinary variables or structure members.

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
}POINT;
void display(int, int);
main()
{
    POINT p1 = {2, 3};
    display(p1.x, p1.y);
    return 0;
}
void display( int a, int b)
{
    printf(" The coordinates of the point are: %d %d", a, b);
}
```

OUTPUT:

The coordinates of the point are: 2 3

PASSING A STRUCTURE TO A FUNCTION

- ✿ When a structure is passed as an argument, **it is passed using call by value method. That is a copy of each member of the structure is made. No doubt, this is a very inefficient method especially when the structure is very big or the function is called frequently.** Therefore, in such a situation passing and working with pointers may be more efficient.
- ✿ The general syntax for passing a structure to a function and returning a structure can be given as, struct struct_name func_name(struct struct_name struct_var);
- ✿ The code given below passes a structure to the function using call-by-value method.

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
}POINT;
void display(POINT);
main()
{
    POINT p1 = {2, 3};
    display(p1);
    return 0;
}
void display( POINT p)
{
    printf(" The coordinates of the point are: %d %d", p.x, p.y);
}
```

OUTPUT:

The coordinates of the point are: 2 3

Guided activity on structures and functions

```
struct fraction {
    int numerator ;
    int denominator ;
};

void show ( struct fraction f )
{
    printf ( " %d / %d ", f.numerator,
            f.denominator );
}

int main ( ) {
    struct fraction f1 = { 7, 12 };
    show ( f1 ) ;
}
```

OUTPUT:

7 / 12

PASSING STRUCTURES THROUGH POINTERS

C allows to create a pointer to a structure. Like in other cases, a pointer to a structure **is never itself a structure, but merely a variable that holds the address of a structure**. The syntax to declare a pointer to a structure can be given as

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .....
}*ptr;
```

OR

```
struct struct_name *ptr;
```

For our student structure we **can declare a pointer variable** by writing

```
struct student *ptr_stud, stud;
```

The next step is to **assign the address of stud to the pointer using the address operator (&)**. So to assign the address, we will write

```
ptr_stud = &stud;
```

To **access the members of the structure**, one way is to write

```
/* get the structure, then select a member */
(*ptr_stud).roll_no;
```

An alternative to the above statement can be used by using 'pointing-to' operator (->) as shown below.

```
/* the roll_no in the structure ptr_stud points to */
ptr_stud->roll_no = 01;
```

The selection operator (->) is a single token, so do not place any white space between them.

Write a program using pointer to structure to initialize the members in the structure

```
#include<stdio.h>
#include<string.h>
struct student
{
    int r_no;
    char name[20];
    char course[20];
    float fees;
};
main()
{
    struct student stud1, *ptr_stud1;
    ptr_stud1 = &stud1;
    ptr_stud1->r_no = 01;
    strcpy(ptr_stud1->name, "Rahul");
    strcpy(ptr_stud1->course, "BCA");
    ptr_stud1->fees = 45000;
    printf("\n DETAILS OF STUDENT");
    printf("\n ----- ");
    printf("\n ROLL NUMBER = %d", ptr_stud1->r_no);
    printf("\n NAME =", puts(ptr_stud1->name));
    printf("\n COURSE = ", puts(ptr_stud1->course));
    printf("\n FEES = %f", ptr_stud1->fees);
}
```

OUTPUT:

DETAILS OF STUDENT

ROLL NUMBER = 1

NAME = Rahul

COURSE = BCA

FEES = 45000.000000

Guided activity on Pointer to a structure

```
struct product
{
    int prodid;
    char name[20];
};
int main()
{
    struct product inventory[3];
    struct product *ptr;
    printf("Read Product Details : \n");
    for(ptr = inventory;ptr<inventory +3;ptr++) {
        scanf("%d %s", &ptr->prodid, ptr->name);
    }
    printf("\noutput\n");
    for(ptr=inventory;ptr<inventory+3;ptr++)
    {
        printf("\n\nProduct ID :%5d",ptr->prodid);
        printf("\nName          : %s",ptr->name);
    }
}
```

Accessing structure members through pointer :

i) Using . (dot) operator :

```
( *ptr ) . prodid = 111 ;
strcpy ( ( *ptr ) . Name, "Pen" ) ;
```

ii) Using - > (arrow) operator :

```
ptr - > prodid = 111 ;
strcpy( ptr->name , "Pencil" ) ;
```

Read Product Details :

111 Pen
112 Pencil
113 Book

Print Product Details :

Product ID : 111
Name : Pen
Product ID : 112
Name : Pencil
Product ID : 113
Name : Book

5.4 SELF REFERENTIAL STRUCTURES

A self referential structure is one that includes at least one member which is a pointer to the same structure type. With self referential structures, we can create very useful data structures such as linked -lists, trees and graphs.

Self referential structures are those structures that contain a reference to data of its same type. That is, a self referential structure in addition to other data contains a pointer to a data that is of the same type as that of the structure. For example, consider the structure node given below.

```
struct node
{
    int val;
    struct node *next;
};
```

Here the structure node will contain two types of data- an integer val and next that is a pointer to a node. You must be wondering why do we need such a structure? Actually, self-referential structure is the foundation of other data structures.

Self referential structures

```
struct student_node {
    int roll_no ;
    char name [25] ;
    struct student_node *next ;
};
int main( )
{
    struct student_node s1 ;
    struct student_node s2 = { 1111, "B.Mahesh", NULL } ;
    s1.roll_no = 1234 ;
    strcpy ( s1.name , "P.Kiran " ) ;

    s1.next = & s2 ;

    printf ( " %s ", s1.name ) ;

    printf ( " %s " , s1.next - > name ) ;
}
```

s2 node is linked to s1
node

Prints P.Kiran

Prints B.Mahesh

OUTPUT:

GUIDED ACTIVITY – Here is the activity you on (self referential structure – foundation for linked list)

```

struct node {
    int rollno; struct node *next;
};
int main() {
    struct node *head,*n1,*n2,*n3,*n4;
    /* creating a new node */
    n1=(struct node *) malloc(sizeof(struct node));
    n1->rollno=101;
    n1->next = NULL;
    /* referencing the first node to head pointer */
    head = n1;
    /* creating a new node */
    n2=(struct node *)malloc(sizeof(struct node));
    n2->rollno=102;
    n2->next = NULL;
    /* linking the second node after first node */
    n1->next = n2;
    /* creating a new node */
    n3=(struct node *)malloc(sizeof(struct node));
    n3->rollno=104;
    n3->next=NULL;
    /* linking the third node after second node */
    n2->next = n3;
    /* creating a new node */
    n4=(struct node *)malloc (sizeof (struct node));
    n4->rollno=103;
    n4->next=NULL;
    /* inserting the new node between
    second node and third node */
    n2->next = n4;
    n4->next = n3;

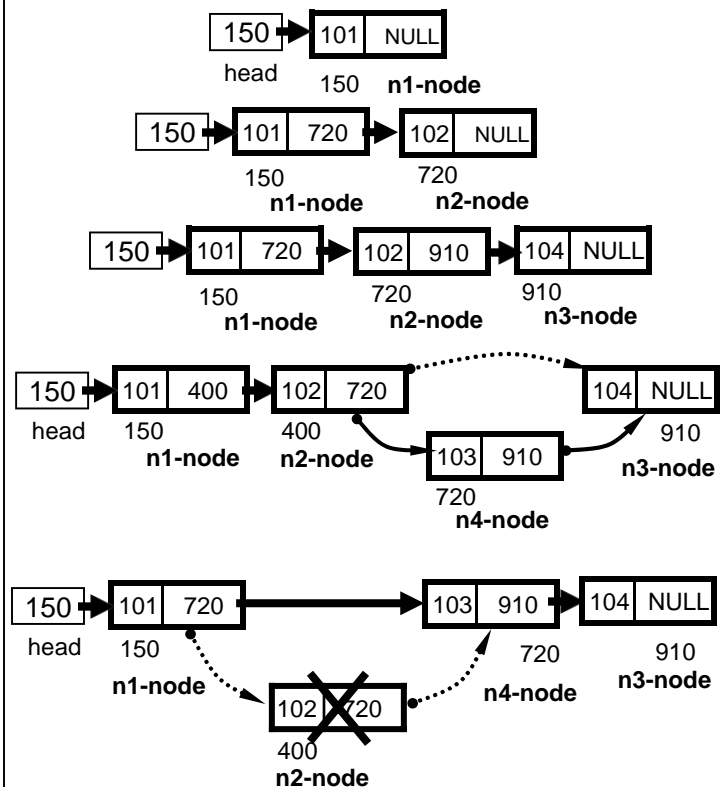
```

Creating a Singly Linked List

```

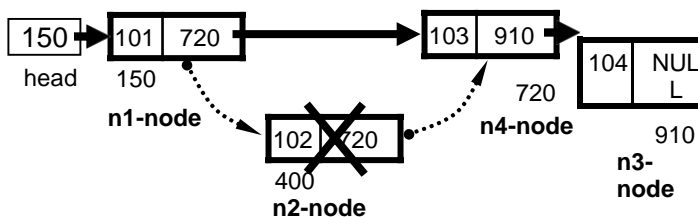
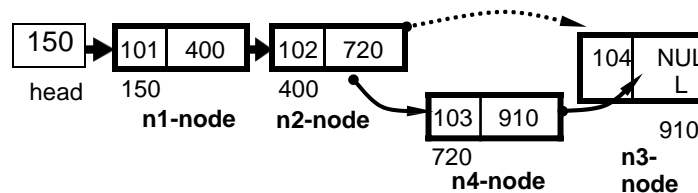
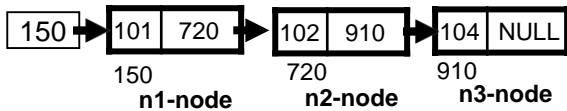
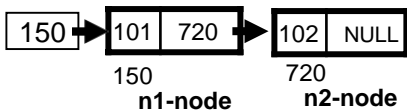
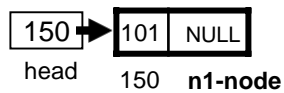
/* deleting n2 node */
n1->next = n4;
free(n2);
}

```



Implementing Singly Linked List

```
struct node *head=NULL;
```



```

#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *next;
};

struct node *createnode() {
    struct node *new;
    new = (struct node *) malloc(sizeof(struct node));
    //printf("\nEnter the data : ");
    //scanf("%d",&new->data);
    new->data=101; //102, 104
    new->next=NULL;
    return new;
}

void append(struct node **h) {
    struct node *new,*temp;
    new = createnode();
    if(*h == NULL) {
        *h = new;
        return;
    }
    temp = *h;
    while(temp->next!=NULL)
        temp = temp->next;
    temp->next = new;
}

void display(struct node *p) {
    printf("\nContents of the List : \n\n");
    while(p!=NULL) {
        printf("\t%d",p->data);
        p = p->next; } }

int main() {
    struct node *head=NULL;
    append(&head);
    display(head);
    append(&head);
    display(head);
    append(&head);
    display(head);
}
  
```

Implementing Singly Linked List

```
struct node {
    int data;
    struct node *next;
};
struct node *createnode() {
    struct node *new;
    new = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter the data : ");
    scanf("%d",&new->data);
    new->next = NULL;
    return new;
}
void append(struct node **h) {
    struct node *new,*temp;
    new = createnode();
    if(*h == NULL) {
        *h = new;
        return;
    }
    temp = *h;
    while(temp->next!=NULL) temp = temp->next;
    temp->next = new;
}
void display(struct node *p) {
    printf("\nContents of the List : \n\n");
    while(p!=NULL) {
        printf("\t%d",p->data);
        p = p->next; } }
```

```
int main() {
    struct node *head=NULL;
    int ch;
    while(1) {
        printf("\n1.Append");
        printf("\n2.Display All");

        printf("\n8.Exit program");
        printf("\n\n\tEnter your choice : ");
        scanf("%d",&ch);
        switch(ch) {
            case 1:append(&head);break;
            case 2:display(head);break;

            ;

            case 8:exit(0);break;
            default :
                printf( "Wrong Choice, Enter correct one : ");
        }
    }
}
```

Compute the age of a person using structure and functions (passing a structure to a function) –Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

Topic. 5.5

Exercise programs:

Compute the age of a person using structure and functions (passing a structure to a function) –

Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

GUIDED ACTIVITY – Here is the activity you on (compare dates in C)

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

```
struct date {  
int dd, mm, yy;} ;
```

```
int date_cmp(struct date d1, struct date d2);  
void date_print(struct date d);
```

```
int main(){  
    struct date d1 = {7, 3, 2005};  
    struct date d2 = {24, 10, 2005};  
    date_print(d1);  
    int cmp = date_cmp(d1, d2);  
    if (cmp == 0)  
        printf(" is equal to");  
    else if (cmp > 0)  
        printf(" is greater i.e. later than ");  
    else printf(" is smaller i.e. earlier than");  
    date_print(d2);  
    return 0;}
```

```
/* compare given dates d1 and d2 */
```

```
int date_cmp(struct date d1, struct date d2){  
    if (d1.dd == d2.dd && d1.mm == d2.mm && d1.yy == d2.yy)  
        return 0;  
    else if (d1.yy > d2.yy || d1.yy == d2.yy && d1.mm > d2.mm || d1.yy == d2.yy  
&& d1.mm == d2.mm && d1.dd > d2.dd)  
        return 1;  
    else return -1;}
```

```
/* print a given date */
```

```
void date_print(struct date d)  {  
    printf("%d/%d/%d", d.dd, d.mm, d.yy);}
```

Compute the age of a person using structure and functions (passing a structure to a function) –

Age Calculator: This program will read your date of birth and print the current age. The logic is behind to implement this program - Program will compare given date with the current date and print how old are you?

```
/*Age Calculator (C program to calculate age).*/
#include<stdio.h>
int date_diff(struct date dt1, struct date dt2);

struct date {
int day, month, year; };

int main() {
    struct date dt1 = {05, 10, 2020};
    struct date dt2 = {17, 05, 2004};
    int cmp = date_diff(dt1, dt2);
    return cmp;}

int date_diff(struct date dt1, struct date dt2){
    int years,months,days;
    if(dt2.year>dt1.year) {
        years=0; months=0; days=0;
        printf("\n I can't travel in time");}
    else if(dt2.year==dt1.year){
        years=0;
        if(dt2.month>dt1.month){
            months=0; days=0;
            printf("\n I can't travel in time");}
        else if(dt2.month==dt1.month){ months=0;
            if(dt2.day>dt1.day){
                days=0;
                printf("\n I can't travel in time");}
            else if(dt2.day==dt1.day){ days=0;
                printf("\n Welcome to Earth");}
            else
                days=dt1.day-dt2.day;}
        else{
            months=dt1.month-dt2.month;
            if(dt2.day>dt1.day) { months--;
                days=30-dt2.day+dt1.day; }
            else
                days=dt1.day-dt2.day;} }
    else {
        years=dt1.year-dt2.year;
        if(dt2.month>dt1.month) {
            years--;
            months=12-dt2.month+dt1.month;
            days=30-dt2.day+dt1.day;}
        else {
            months=dt1.month-dt2.month;
            if(dt2.day>dt1.day) {
                months--;
                days=30-dt2.day+dt1.day; }
            else
                days=dt1.day-dt2.day;} }
    printf("\n Your age is %d years, %d months, %d days",years,months,days);}
```

Your age is 16 years, 4 months, 18 days

GUIDED ACTIVITY – Here is the activity you on (Time in C)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Print current date and time in C
int main(void){
    // variables to store date and time components
    int hours, minutes, seconds, day, month, year;
    // time_t is arithmetic time type
    time_t now;
    // Obtain current time
    // time() returns the current time of the system as a time_t value
    time(&now);
    // Convert to local time format and print to stdout
    printf("Today is : %s", ctime(&now));
    // localtime converts a time_t value to calendar time and
    // returns a pointer to a tm structure with its members
    // filled with the corresponding values
    struct tm *local = localtime(&now);

    hours = local->tm_hour;           // get hours since midnight (0-23)
    minutes = local->tm_min;           // get minutes passed after the hour (0-59)
    seconds = local->tm_sec;           // get seconds passed after minute (0-59)

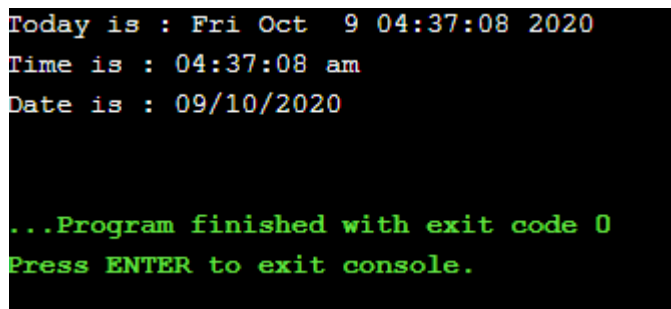
    day = local->tm_mday;              // get day of month (1 to 31)
    month = local->tm_mon + 1;         // get month of year (0 to 11)
    year = local->tm_year + 1900;      // get year since 1900

    // print local time
    if (hours < 12)                   // before midday
        printf("Time is : %02d:%02d:%02d am\n", hours, minutes, seconds);

    else                             // after midday
        printf("Time is : %02d:%02d:%02d pm\n", hours - 12, minutes, seconds);

    // print current date
    printf("Date is : %02d/%02d/%d\n", day, month, year);

    return 0;
}
```



```
Today is : Fri Oct  9 04:37:08 2020
Time is : 04:37:08 am
Date is : 09/10/2020
```

```
...Program finished with exit code 0
Press ENTER to exit console.
```


Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and functions)

```
#include <stdio.h>
#include <time.h>
struct student{
    char lastName[100];
    char firstName[100];
    char *date;
    int age;
    int id;};

int main(){
    int n=1;
    struct student s[n];
    int x;
    do{
        printf("main menu :\n1.add\n2.delete\n3.diplay\n4.exit\n");
        scanf("%d",&x);
        switch(x){
            case 1:
                for(int i=0;i<n;i++){
                    printf("Enter first name\n");
                    scanf("%s",s[i].firstName);
                    printf("Enter last name\n");
                    scanf("%s",s[i].lastName);
                    printf("Enter your id\n");
                    scanf("%d",&s[i].time);
                    printf("Enter your age\n");
                    scanf("%d",&s[i].age);
                    time_t timer;
                    timer=time(NULL);
                    s[i].date = asctime(localtime(&timer));
                    //s[i].time=time(&now);
                }
                for(int i=0;i<n;i++){
                    printf("id\tfirstName\tlastName\tage\tdate\n%d\t%s\t%s\t%d\t%s",s[i].id,s[i].firstNa
me,s[i].lastName,s[i].age,s[i].date);
                }
                break;
            case 2:
                break;
            case 3:
                break;
            case 4:
                break;
            default:
                printf("wrong choice");
                break;
        }
    }while(x!=4);
    return 0;
}

Note:    time_t t;
         time(&t);
         printf("\n current time is : %s",ctime(&t));
```

Quiz



Test Yourself –5.1 to 5.6 Topics (quiz)

- 1) A data structure that can store related information together is called
(a) Array (b) String (c) Structure (d) All of these
- 2) A data structure that can store related information of different data types together is called
(a) Array (b) String (c) Structure (d) All of these
- 3) Memory for a structure is allocated at the time of
Structure definition Structure variable declaration
Structure declaration Function declaration
- 4) A structure member variable is generally accessed using
(a) Address operator (b) Dot operator
(c) Comma operator (d) Ternary operator
- 5) A structure that can be placed within another structure is known as
Self-referential structure Nested structure
Parallel structure Pointer to structure
- 6) A union member variable is generally accessed using the
(a) Address operator (b) Dot operator
(c) Comma operator (d) Ternary operator
- 7) typedef can be used with which of these data types?
(a) struct (b) union
(c) enum (d) all of these

Assignment

Unit V

Assignment Questions

CO 1	Develop C program solutions to simple computational problems		
1.	<p>Declare a structure that represents the following hierarchical information.</p> <ul style="list-style-type: none">Student<ul style="list-style-type: none">Roll NumberName<ul style="list-style-type: none">First nameMiddle NameLast NameSexDate of Birth<ul style="list-style-type: none">DayMonthYearMarks<ul style="list-style-type: none">EnglishMathematicsComputer Science	K2	CO1
2.	<p>Define a structure date containing three integers— day, month, and year. Write a program using functions to read data, to validate the date entered by the user and then print the date on the screen. For example, if you enter 29,2,2010 then that is an invalid date as 2010 is not a leap year. Similarly 31,6,2007 is invalid as June does not have 31 days.</p>	K2	CO1
3.	<p>Write a program to define a union and a structure both having exactly the same members. Using the sizeof operator, print the size of structure variable as well as union variable and comment on the result.</p>	K2	CO1

Part A

Question & Answer

Part A

1. What is Structure? Write the syntax for structure.	K3	CO3	S
2. How the members of structure object is accessed?	K3	CO3	S
3. What is a nested structure?	K3	CO3	S
4. How typedef is used in structure?	K3	CO3	A
5. What is meant by Self-referential structures?	K3	CO3	A
6. Develop a structure namely Book and create array of Book structure with size of ten.	K2	CO3	S
7. Invent the application of size of operator to this structure. Consider the declaration: struct { char name; int num; } student;	K2	CO3	S
8. List the use of typedef.	K2	CO3	A
9. Differentiate between Structure and Array.	K2	CO3	A
10. Define the meaning of Array structure.	K2	CO3	A

Part B

Questions

Part B

1. Describe about the functions and structures. (13)	K3	CO3	S
2. Explain about the structures and its operations with example programs (13)	K3	CO3	S
3. Explain about array of structures and nested structures with example.(13)	K3	CO3	A
4. Write a C program using structures to prepare the students mark statement. (13)	K3	CO3	A
5. Write a C program using structures to prepare the employee payroll. (13)	K3	CO3	A
6. Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and function)	K3	CO3	S

Supportive Online Certification

Unit V

Certification Courses

⚙ NPTEL

Problem solving through Programming in C

<https://nptel.ac.in/courses/106/105/106105171/>

⚙ Coursera

1) C for Everyone: Structured Programming

<https://www.coursera.org/learn/c-structured-programming>

2) C for Everyone: Programming Fundamentals

<https://www.coursera.org/learn/c-for-everyone>

Real time Applications

Unit V

Functions are used at many places in real life applications and some applications are listed here.

- ✿ commonly used in calculation or computation purpose

- ✿ Major imaging activity

Content beyond syllabus

Unit V

Content beyond syllabus

1) Comparison between structure and union

Assessment Schedule

Unit V

Prescribed Text book & References

Unit V

Text books & References

TEXT BOOK

1. Reema Thareja, "Programming in C", Oxford University Press, Second Edition, 2016

REFERENCES:

1. Kernighan, B.W and Ritchie,D.M, "The C Programming language", Second Edition, Pearson Education, 2006
2. Paul Deitel and Harvey Deitel, "C How to Program", Seventh edition, Pearson Publication
3. Juneja, B. L and Anita Seth, "Programming in C", CENGAGE Learning India pvt. Ltd., 2011
4. Pradip Dey, Manas Ghosh, "Fundamentals of Computing and Programming in C", First Edition, Oxford University Press, 2009
5. ER and ETA , "CC Foundation Program Reference materials" Infosys Ltd.

Mini Project Suggestions

Unit V

- 1) Scientific calculator using strcture
- 2) Employee Management System

Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.