

*PROBLEM
SOLVING
THROUGH
C*

<u>Contents</u>	<u>Page Number</u>
Unit - I	03
Unit - II	37
Unit - III	66
Unit - IV	83
Unit - V	108

Unit – I

Introduction to Computers : Hardware components : CPU, RAM, I/O Devices.

Software : Application software and system Software, Computer Languages, Simple Problem solving : Algorithms, Pseudo code, Flow charts. Example Problems – Solving quadratic equation, finding sum of digits of a given number.

Introduction to c Language : Structure of c program, Compilation process. Basic Programming constructs – Identifiers, Basic Data types, Variables, Constants, I/O functions, Operators – Expressions, Precedence and Associativity, Expression Evaluation, Type conversion, Bitwise operators, Statements, Programming examples.

Introduction to Computer

All types of computers follow a same basic logical structure and perform the following three basic operations for converting input data into useful information.

Sr.No.	Operation	Description
1	Take Input Data	The process of entering data and instructions into the computer system
2	Processing Data	Performing arithmetic, and logical operations on data in order to convert them into useful information.
3	Output Information	The process of producing useful information or results for the user.

Hardware and software are mutually dependent on each other. Both of them must work together to make a computer produce a useful output.

- Software cannot be utilized without supporting hardware.
- Hardware without set of programs cannot be utilized and is useless.
- Different software applications can be loaded on a hardware for different purposes.
- A software acts as an interface between the user and the hardware.

Hardware Components:

Hardware represents the physical components of a computer i.e. the components that can be seen and touched.

Without any hardware, computer would not exist, and software could not be used.

Examples of Hardware are following:

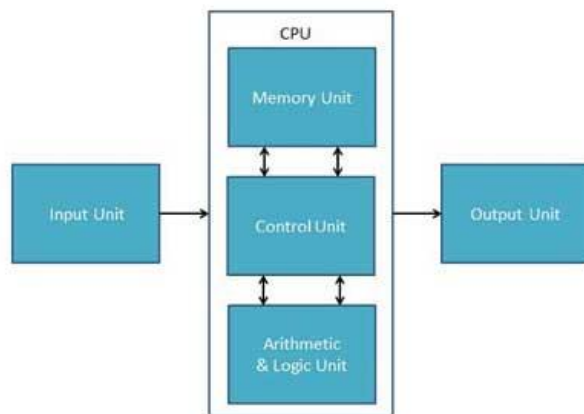
- Input devices -- keyboard, mouse etc.
- Output devices -- printer, monitor etc.
- Secondary storage devices -- Hard disk, CD, DVD etc.
- Internal components -- CPU, motherboard, RAM etc.

CPU (Central Processing Unit)

CPU is considered as the brain of the computer. CPU performs all types of data processing operations. It stores data, intermediate results and instructions(program). It controls the operation of all parts of computer.

CPU itself has following three components

- Memory Unit
- Control Unit
- ALU(Arithmetic Logic Unit)



Memory Unit

This unit can store instructions, data. This unit passes information to the other units of the computer when needed. It is also known as main memory or primary storage or Random access memory(RAM). As soon as the machine is switched off, data is erased in RAM. RAM is volatile,. RAM is small, both in terms of its physical size and in the amount of data it can hold.It stores final results of processing before these results are released to an output device.All inputs and outputs are transmitted through main memory.

Control Unit

This unit controls the operations of all parts of computer but does not carry out any actual data processing operations.

Functions of this unit are:

- It is responsible for controlling the transfer of data and instructions among other units of a computer.
- It manages and coordinates all the units of the computer.
- It obtains the instructions from the memory, interprets them, and directs the operation of the computer.
- It communicates with Input/Output devices for transfer of data or results from storage.
- It does not process or store data.

ALU(Arithmetic Logic Unit)

This unit consists of two subsections namely

- Arithmetic section
- Logic Section

Arithmetic Unit

Function of arithmetic section is to perform arithmetic operations like addition, subtraction, multiplication and division. Any complex operations are done by making repetitive use of above operations only.

Logic Unit

Function of logic section is to perform logic operations such as comparing or matching of data.

Input Devices

Following are few of the important input devices which are used in a computer:

- Keyboard
- Mouse
- Joy Stick
- Light pen
- Scanner
- Graphic Tablet
- Microphone
- Magnetic Ink Card Reader(MICR)
- Optical Character Reader(OCR)
- Bar Code Reader
- Optical Mark Reader(OMR)

Output Devices

Following are few of the important output devices which are used in a computer.

- Monitors
- Graphic Plotter
- Printer

Software

System software and application software are the two main types of computer software.

Application Software

Application software is designed to satisfy a particular need. Application software may consist of a single program, such as a Microsoft's notepad for writing and editing simple text. It may also consist of a collection of programs, often called a software package, which work together to accomplish a task, such as a spreadsheet package.

Examples of Application software are following:

- Payroll Software
- Student Information Software
- Inventory Management Software
- Income Tax Software
- Railways Reservation Software
- Library Information System
- College Information System

System Software

System software is a type of computer program that is designed to run a computer's hardware and application programs. The system software is collection of programs designed to operate, control, and extend the processing capabilities of the computer itself. System software serves as the interface between hardware and the end users.

Some examples of system software are Operating System, Compilers, Interpreter, Assemblers etc.

Other examples of system software and what each does:

- The BIOS (basic input/output system) gets the computer system started after you turn it on and manages the data flow between the operating system and attached devices such as the hard disk, video adapter, keyboard, mouse, and printer.
- The boot program loads the operating system into the computer's main memory or random access memory (RAM).
- A device driver controls a particular type of device that is attached to computer, such as a printer or scanner. The driver program converts the more general input/output instructions of the operating system into the form that the device can understand.

Computer Languages

Different Computer language available and these various languages are for expressing a set of instructions for a digital computer. Such instructions can be executed directly when they are in ones and zeros form known as machine language, after a simple substitution process when expressed in a corresponding assembly language, or after translation from some “higher-level” language. Although there are over 2,000 computer languages, relatively few are widely used.

Machine Language

A machine language consists of the numeric codes for the operations that a particular computer can execute directly. The codes are strings of 0s and 1s. The symbol 0 stands for the 0 volts and the 1 stands for an electric pulse generally it is 5 volts. Machine language is the lowest and most elementary level of programming language and was the first type of programming language to be developed. Machine language is basically the only language that a computer can understand.

Machine language instructions typically use some bits to represent operations, such as addition, and some to represent operands. Machine language is difficult to read and write, since it does not resemble conventional mathematical notation or human language, and its codes vary from computer to computer.

Advantages

Machine language makes fast and efficient use of the computer.

It requires no translator of the code. It is directly understood by the computer.

Disadvantages

All operation codes have to be remembered

All memory addresses have to be remembered.

Advantages

Disadvantages

It is hard to find errors in a program written in the machine language.

Assembly Language

Assembly language was developed to overcome some of the disadvantages of machine language. This is another low-level but very important language in which operation codes and operands are given in the form of alphanumeric symbols instead of 0's and 1's. These alphanumeric codes are known as mnemonic codes e.g. ADD for addition, SUB for subtraction, START etc. Because of this feature, assembly language is also known as 'Symbolic Programming Language.'

This language is also very difficult and needs a lot of practice to master it because there is only a little English support in this language. The instructions of the assembly language are converted to machine codes by an assembler.

Advantages

Disadvantages

Assembly language is easier to understand and use as compared to machine language.

Like machine language, it is also machine dependent.

It is easy to locate and correct errors. Easily modified

Since it is machine dependent, the programmer also needs to understand the hardware.

High-Level Languages

High-level computer languages look similar to English. The purpose of developing high-level languages was to enable people to write programs easily, in their own native language environment (English).

High-level languages are basically symbolic languages that use English words and mathematical symbols rather than mnemonic codes. Each instruction in the high-level language is translated into machine language instructions that the computer can understand by compiler or interpreter.

Advantages

Disadvantages

High-level languages are user-friendly

A high-level language has to be translated into the machine language by a translator

Advantages	Disadvantages
They are similar to English and use English vocabulary and well-known symbols	
They are easier to learn	
They are easier to maintain	
A program written in a high-level language can run on any computer	

Algorithm

Algorithms were originally born as part of mathematics but currently the word is strongly associated with computer science, an algorithm is a procedure or formula for solving a problem. In mathematics and computer science, an algorithm usually means a small procedure that solves a problem.

An algorithm specifies a series of steps that perform a particular computation or task.

- An algorithm expects a defined set of inputs.
- An algorithm produces a defined set of outputs.
- An algorithm is guaranteed to terminate and produce a result, always stopping after a finite time.

An algorithm is a procedure for solving a problem in terms of the actions to be executed and the order in which those actions are to be executed. An algorithm is merely the sequence of steps taken to solve a problem. The steps are normally "sequence," "selection," "iteration," and a case-type statement.

Pseudo-code

Pseudo-code is an informal way to express the design of a computer program or an algorithm. The aim is to get the idea quickly and also easy to read without details. An algorithm is a systematic logical approach used to solve problems in a computer while Pseudo-code is the statement in plain English which may be translated later into a program. Pseudo code is an intermediary between an algorithm and program.

Pseudo-code is not actual programming language. It uses short phrases to write code for programs before actually create it in a specific language.

A Pseudo-code Examples

1)Task to add 2 numbers together and then display the result.

Enter two numbers, A, B

set sum to A+B

Print Sum

2)Area of a rectangle:

Read l, w

Compute the area = l*w



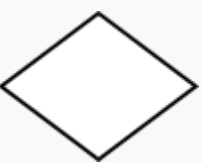

Display the area

3)pseudo-code tocompute the perimeter of a rectangle:

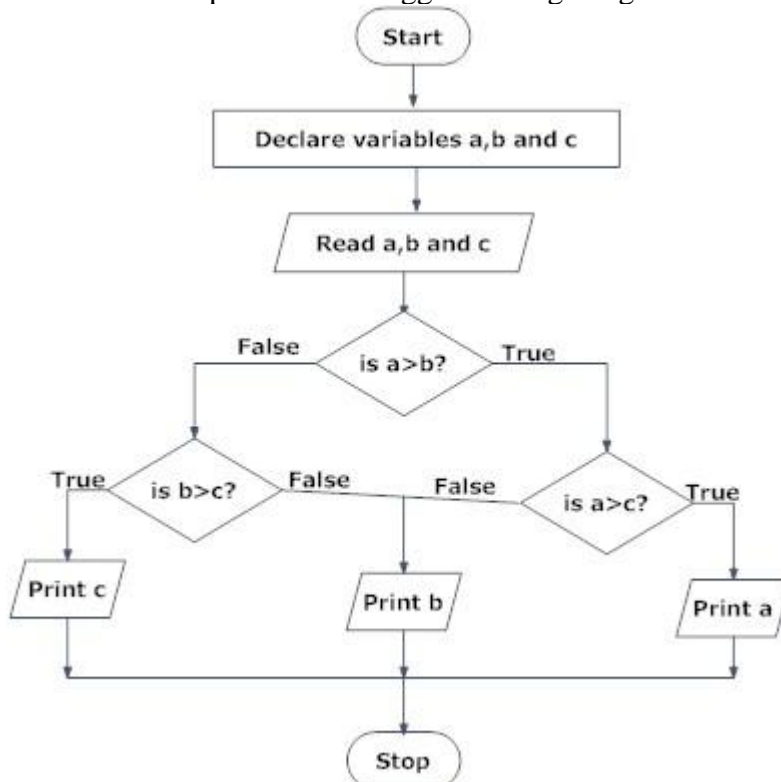
Read l
 Read w
 Compute Perimeter = $2*(l + w)$
 Display Perimeter

Flowchart

A flowchart is a type of diagram that represents an algorithm, workflow or process, showing the steps as symbols of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution to a given problem.

	Start/Stop	The ovals, or rounded rectangles. They usually contain the word "Start" or "Stop".
	Process	Represented as rectangles. This shape is used to show Arithmetic operation is performed. Examples: "Add 1 to X", $x=x+5$, $r= n\%2$ etc
	Decision	Represented as a diamond (rhombus) showing where a decision is necessary, commonly a Yes/No (True/False)condition. The conditional symbol has two arrows coming out of it, one corresponding to Yes and one corresponding to No.
	Input/Output	Represented as a parallelogram. Involves receiving data and displaying output

Flowchart example : To find biggest among the given three numbers



Task : Roots of a quadratic equation

Algorithm

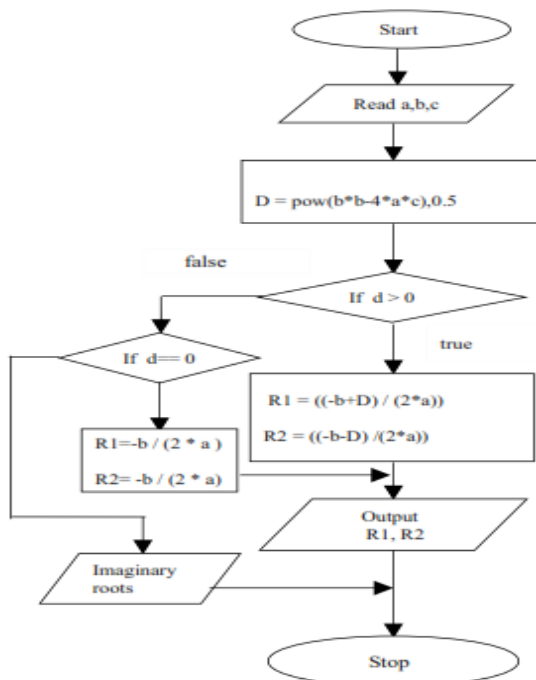
1. Read coefficient a, b, c
2. Calculate $dis = b^2 - 4 * a * c$
3. If dis is less than 0 then roots are imaginary. If dis is 0 then print $-b/2a$ as a root. Otherwise print $(-b + \sqrt{dis})/2 * a$ and $(-b - \sqrt{dis})/2 * a$ are two real roots.

Pseudo-code

```

Step 1: Start
Step 2: Read A, B, C as integer
Step 3: Declare disc, x1, x2 as float
Step 4: Assign dis = (B * B) - (4 * A * C)
Step 5: if( dis > 0 )
begin
    Print "THE ROOTS ARE REAL ROOTS"
    Set x1 ← ((-B) + (sqrt(dis)) / 2 * A)
    Set x2 ← ((-B) - (sqrt(dis)) / 2 * A)
    Print x1, x2
end
else if( dis = 0)
begin
    Print " THE ROOTS ARE SAME"
    Assign x1 ← -B / 2 * A
    Print x1
end
else Print "THE ROOTS ARE IMAGINARY ROOTS"
Step 6 : stop
    
```

Flowchart



Task : Sum of digits of a given number

Algorithm

Input a Number

- Initialize Sum to zero
- While Number is not zero
 - Get Remainder by Number Mod 10
 - Add Remainder to Sum
 - Divide Number by 10
- Print sum

Pseudo-code

Step 1: Input N

Step 2: Sum = 0

Step 3: While (N != 0)

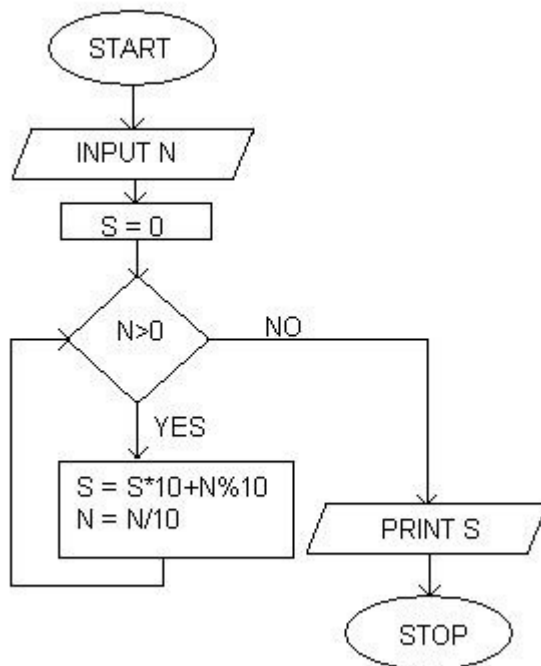
Rem = N % 10;

Sum = Sum + Rem;

N = N / 10;

Step 4: Print Sum

Flowchart



Structure of c Program

Documentation Section
Preprocessor Directive Section
Global Declaration Section
<pre> main() Function Section { Declaration Part Executable Part } </pre>
User Defined Function

Documentation section

We can write comments about the program, creation or modified date, author name etc in this section. The characters or words or anything which are written in between “/*” and “*/” or after //. These will be ignored by C compiler during compilation. Documentation section helps anyone to know an overview of the program.

Example : /* Program to find the average of two numbers */

Preprocessor directives Section

All the symbolic constants are written in this section. Macros are known as symbolic constants. Header files that are required to execute a C program are included in this section

Global declaration section

Global variables are defined in this section. When a variable is to be used throughout the program in two or more functions, can be defined in this section.

Main function

It is necessary to have one main() function in C program. C program is started from main function. main() function contains two major sections called declaration section and executable section. The declaration section declares all the variables that are used in executable section. These two parts must be written in between the opening and closing braces. Each statement in the declaration and executable sections must end with a semicolon (;).

Use defined functions

User can write their own functions in this section which perform a particular task.

Example 1. First c program

```

/* First Hello World program */
#include <stdio.h>

int main()

```

```
{
printf(" Hello World");
return 0;
}
```

Let us take a look at the various parts of the above program –

The First line `/*...*/` is a comment and will be ignored by the compiler.

The nextline of the program `#include<stdio.h>` is a preprocessor command, which tells a C compiler to include `stdio.h` file before going to actual compilation.

The next line `intmain()` is the main function where the program execution begins.

- The open brace indicates the beginning of a block of code.
- The next line `printf(...)` is another function for the message "Hello, World!" to be displayed on the monitor.
- The next line `return 0;` terminates the `main()` function and returns the value 0

Compilation Process

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps –

Open a text editor and type the code.

Save the file.

- Open a command prompt and go to the directory where you have saved the file.
- Type `gccfilename` and press enter to compile code.
- If there are no errors in the code, the command prompt will take you to the next line and would generate `a.out` executable file.
- Now, type `./a.out` to execute your program.
- You will see the output "Hello World" printed on the monitor.

C Tokens

C tokens are the basic buildings blocks in C language which are constructed together to write a C program.Each and every smallest individual unit in a C program are known as C tokens.

- C tokens are

1. Keywords (eg: `int`, `while`),
2. Identifiers (eg: `main`, `total`),
3. Constants (eg: `10`, `20`,`23.5`),
4. Strings (eg: `"total"`, `"hello"`),
5. Special symbols (eg: `()`, `{}`),
6. Operators (eg: `+`, `/`,`-`,`*`)

Identifiers

A c identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore `'_'` followed by letters, and digits (0 to 9).C does not allow punctuation characters such as `@`, `$`, and `%` within identifiers. C is a case-sensitive programming language.

Identifiers are case sensitive, so "NUMBER", "number", and "Number" are three different identifiers.

Key Words

Keywords are pre-defined words in a C compiler. Each keyword is meant for a specific purpose in a C program. Since keywords can't be used as variable name.

C language supports 32 keywords which are given below

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

Data Types

C has a concept of 'data types' which are used to define a variable before it is used. The definition of a variable will assign memory location for the variable and define the type of data that will be stored in the location.

C has the following basic datatypes.

- int
- float
- double
- char

Modifiers

The data types mentioned above have the following modifiers.

- short
- long
- signed
- unsigned

Integer Data Type

- Integer data type allows a variable to store numeric values.

- “int” keyword is used to refer integer data type.
- The storage size of int data type is 2 or 4 byte. It varies depend upon the processor in the CPU that we use. If we are using 16 bit processor, 2 byte (16 bit) of memory will be allocated for int data type. Like wise, 4 byte (32 bit) of memory for 32 bit processor and 8 byte (64 bit) of memory for 64 bit processor is allocated for int datatype.
- If you want to use the integer value that crosses the above limit, you can go for “long int” for which the limits are very high.
- If we use int data type to store decimal values, decimal values will be truncated and we will get only whole number.
- In this case, float data type can be used to store decimal values in a variable.

Character Data type

- Character data type allows a variable to store only one character.
- Storage size of character data type is 1. We can store only one character using character data type.
- “char” keyword is used to refer character data type.
- For example, ‘A’ can be stored using char datatype. We can’t store more than one character using char data type.

Floating point Data Type

Floating point data type consists of 2 types. They are,

1. float
2. double

Float

- Float data type allows a variable to store decimal values.
- Storage size of float data type is 4. This also varies depend upon the processor in the CPU.
- We can use up-to 6 digits after decimal using float data type.
- For example, 10.456789 can be stored in a variable using float data type.

Double

- Double data type is also same as float data type which allows up-to 15 digits after decimal.

Type	Bytes	Range	Format specifier
Char	1	128 to 127	%c
short int	2	-32,768 to 32,767	%hi
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to +2,147,483,647	%d
long int	8	-9223372036854775808 to 9223372036854775807	%li
unsigned long int	8	0 to 18446744073709551615	%lu
float	4	max : 3.408E38 to 1.17E-38 min : -3.408E38 to -1.17E-38	%f
double	8	2.7E-308 to 1.7E308 15 decimal places precision	%lf
long double	16	3.4E-4932 to 1.1E4932 19 decimal places precision	%Lf

Variable

- A variable is nothing but a name given to a storage area that our programs can manipulate.
- Each variable in C has a specific type, which determines the size and the range of values that can be stored within the memory. The set of operations that can be applied to the variable.
- The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

Some valid declarations are shown here –

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

Variables can be initialized (assigned an initial value) at the time of declaration. The initialization consists of an equal sign followed by a constant expression follows –

Some examples are –

```
int d = 3, f = 5;           // definition and initializing d and f.
float z = 22.6;           // definition and initializes z.
char x = 'p';             // the character variable x has the value 'p'.
```

For definition without an initialization, the initial value of the variables are undefined.

Lvalues and Rvalues in C

There are two kinds of expressions in C –

- lvalue – Expressions that refer to a memory location are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- rvalue – An rvalue may appear on the right-hand side but not on the left-hand side of an assignment.

Variables are lvalues and so they may appear on the left-hand side of an assignment. Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side. Take a look at the following valid and invalid statements –

```
g = 20; // valid statement
10 = g; // invalid statement; would generate compile-time error
```

Constants

- Constant variables are also like normal variables. But, only difference is, their values can not be modified by the program once they are defined.
- Constants refer to fixed values. They are also called as literals

Syntax:


```
const data_type variable_name;
    const double PI = 3.14;
```

Here, PI is a constant variable.

Types of c Constants

1. Integer constants
2. Real or Floating point constants
3. Octal & Hexadecimal constants
4. Character constants
5. String constants
6. Backslash character constants

S.no	Constant type	data type	Example
1	Integer constants	unsigned long int short int int	53, 762, -478, 5000u, 1000U, 483,647
2	Real or Floating point constants	float double	10.456789, 0.02-E2 0.123456789
3	Octal constant	Int	013 /* starts with 0 */
4	Hexadecimal constant	Int	0x90 /* starts with 0x */
5	character constants	char	'A' , '+', '1'
6	string constants	char	"ABCD" , "Hai"

Rules for constructing c Constants

Integer Constants

- An integer constant must have at least one digit.
- It must not have a decimal point.
- It can either be positive or negative.
- No commas or blanks are allowed within an integer constant.
- If no sign precedes an integer constant, it is assumed to be positive.

Real Constants

- A real constant must have at least one digit
- It must have a decimal point
- It could be either positive or negative
- If no sign precedes an integer constant, it is assumed to be positive.
- No commas or blanks are allowed within a real constant.

Character constants

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.
- The maximum length of a character constant is 1 character.

String Constants

- String constants are enclosed within double quotes.
Back slash character constants
- There are some characters which have special meaning in C language.
- They should be preceded by backslash symbol (\)
- Given below is the list of special characters and their purpose.

Backslashcharacter/Escape Sequence character	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\\	Backslash
\v	Vertical tab
\a	Alert or bell
\?	Question mark

I/O Functions

Input : In any programming language input means to enter some data into program through keyboard. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

Output : In any programming language output means to display data on screen, printer or in any file. C programming language provides a set of built-in functions to output required data.

printf() function

This is one of the most frequently used functions in C for output.

Try following program to understand printf() function.

```
#include <stdio.h>
main()
```

```

{
intdec = 5;
charch = 's';
float pi = 3.14;

printf("%d,%f, %c\n", dec, pi, ch);
}

```

The output of the above would be:

```
5, 3.140000 ,s
```

Here %d is being used to print an integer, %f is being used to print a float and %c is being used to print a character.

scanf() function

This is the function which can be used to read an input from the command line.

```

#include<stdio.h>
int main()
{
inttestInteger;
printf("Enter an integer: ");
scanf("%d",&testInteger);
printf("Number = %d",testInteger);
return0;
}

```

If we execute the above program following output will be generated

```
Enter an integer: 4
Number = 4
```

Here %d is being used to read an integer value and we are passing &testinteger to store the value read as input. Here & indicates the address of variable. This program will prompt you to enter a value. Whatever integer value we enter at command prompt that will be output at the screen using printf() function.

```

#include<stdio.h>
int main()
{
float f;
printf("Enter a number: ");
// %f format string is used in case of floats
scanf("%f",&f);
printf("Value = %f", f);
return0;
}

```

Output

```
Enter a number: 23.45
Value = 23.450000
```

The format string "%f" is used to read and display formatted in case of floats.

Example: C Character I/O

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

```
int main()
{
char var1;
printf("Enter a character: ");
scanf("%c",&var1);
printf("You entered %c.",var1);
return0;
}
```

Output

```
Enter a character: g
You entered g.
```

Format string %c is used in case of character types.

Little bit on ASCII code

When a character is entered in the above program, the character itself is not stored. Instead a numeric value the ASCII value is stored. And when we displayed that value using "%c" , the entered character is displayed.

Example : C ASCII Code

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

```
int main()
```

```
{
```

```
char var1;
```

```
printf("Enter a character: ");
```

```
scanf("%c",&var1);
```

```
// When %c text format is used, character is displayed in case of character types
```

```
printf("You entered %c.\n",var1);
```

```
// When %d text format is used, integer is displayed in case of character types
```

```
printf("ASCII value of %c is %d.", var1, var1);
```

```
return0;
```

```
}
```

Output

```
Enter a character: g
```

```
You entered g.
```

```
ASCII value of g is 103.
```

The ASCII value of character 'g' is 103. When, 'g' is entered, 103 is stored in variable var1 instead of g.

You can display a character if you know ASCII code of that character. This is shown by following example.

Example: C ASCII Code

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

```
int main()
```

```
{
```

```
int var1 =69;
```

```
printf("Character having ASCII value 69 is %c.",var1);
```

```
return0;
```

```
}
```

Output

Character having ASCII value 69 is E.

More on Input/Output

Integer and floats can be displayed in different formats in C programming.

Example 8: I/O of Floats and Integers

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

```
int main()
```

```
{
```

```
int integer =9876;
```

```
floatdecimal=987.6543;
```

```
// Prints the number right justified within 6 columns
```

```
printf("4 digit integer right justified to 6 column: %6d\n", integer);
```

```
// Tries to print number right justified to 3 digits but the number is not right adjusted because there are only 4 numbers
```

```
printf("4 digit integer right justified to 3 column: %3d\n", integer);
```

```
// Rounds to two digit places
```

```
printf("Floating point number rounded to 2 digits: %.2f\n",decimal);
```

```
// Rounds to 0 digit places
```

```
printf("Floating point number rounded to 0 digits: %.f\n",987.6543);
```

```
// Prints the number in exponential notation(scintific notation)
```

```
printf("Floating point number in exponential form: %e\n",987.6543);
```

```
return0;
```

```
}
```

Output

4 digit integer right justified to 6 column: 9876

4 digit integer right justified to 3 column: 9876

Floating point number rounded to 2 digits: 987.65

Floating point number rounded to 0 digits: 988

Floating point number in exponential form: 9.876543e+02

if- (minus sign) Between % and format specifier it means left justify.

```
printf("%-2.3fn",17.23478);
```

The output on the screen is: 17.235

getchar() &putchar() functions

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

```

Example
#include <stdio.h>
#include <conio.h>
void main( )
{
int c;
printf("Enter a character");
c=getchar();
putchar(c);
return 0;
}

```

Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators

–

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Miscellaneous Operators

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
+	Adds two operands.	A + B = 30
–	Subtracts second operand from the first.	A – B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Example : Arithmetic Operators

```

// C Program to demonstrate the working of arithmetic operators(Binary opearators)
#include<stdio.h>
int main()
{

```

```

int a =9,b =4, c;
    c =a+b;
printf("a+b = %d \n",c);

    c = a-b;
printf("a-b = %d \n",c);

    c = a*b;
printf("a*b = %d \n",c);

    c=a/b;
printf("a/b = %d \n",c);

    c=a%b;
printf("Remainder when a divided by b = %d \n",c);

return0;
}

```

Output

a+b = 13

a-b = 5

a*b = 36

a/b = 2

Remainder when a divided by b=1

In normal calculation, $9/4 = 2.25$. However, the output is 2 in the c program. It is because both operands a and b are integers, the result is also integer. The compiler neglects the term after decimal point and shows answer 2 instead of 2.25.

The modulo operator % computes the remainder. When a = 9 is divided by b = 4, the remainder is 1. The % operator can only be used with integers.

Example : Increment and Decrement Operators

// C Program to demonstrate the working of increment and decrement operators

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

```
int main()
```

```
{
```

```
int a =10, b =100;
```

```
float c =10.5, d =100.5;
```

```
printf("++a = %d \n",++a);
```

```
printf("--b = %d \n",--b);
```

```
printf("++c = %f \n",++c);
```

```
printf("--d = %f \n",--d);
```

```
return0;
```

```
}
```

Output

++a = 11

--b = 99

++c = 11.500000

++d = 99.500000

Here, the operators ++ and -- are used as prefix. These two operators can also be used as postfix like a++ and a--.

if a=10, res= ++a; Here res value is 11 and a value is also 11.(Pre Increment)

if a=10, res= a++; Here res value is 10 and a value is also 11. (Post Increment)

Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0. Relational operators are used in decision making and loops.

The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true value is 0.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true, value is 1 .
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true, value is 0.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true, value is 1 .
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true, value is 0.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true, value is 1.

Example : Relational Operators

// C Program to demonstrate the working of arithmetic operators

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

```
int main()
```

```
{
```

```
int a =5, b =5, c =10;
```

```
printf("%d == %d = %d \n", a, b, a == b);// true
```

```
printf("%d == %d = %d \n", a, c, a == c);// false
```

```
printf("%d > %d = %d \n", a, b, a > b);//false
```

```
printf("%d > %d = %d \n", a, c, a > c);//false
```

```
printf("%d < %d = %d \n", a, b, a < b);//false
```



```
printf("%d < %d = %d \n", a, c, a < c);//true
printf("%d != %d = %d \n", a, b, a != b);//false
printf("%d != %d = %d \n", a, c, a != c);//true
printf("%d >= %d = %d \n", a, b, a >= b);//true
printf("%d >= %d = %d \n", a, c, a >= c);//false
printf("%d <= %d = %d \n", a, b, a <= b);//true
printf("%d <= %d = %d \n", a, c, a <= c);//true

return 0;

}
```

Output

```
5 == 5 = 1
5 == 10 = 0
5 > 5 = 0
5 > 10 = 0
5 < 5 = 0
5 < 10 = 1
5 != 5 = 0
5 != 10 = 1
5 >= 5 = 1
5 >= 10 = 0
5 <= 5 = 1
5 <= 10 = 1
```

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming. Used to combine relational expressions.

Following table shows logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then –

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Example : Logical Operators

// C Program to demonstrate the working of logical operators

```
#include<stdio.h>
int main()
```

```

{
int a =5, b =5, c =10, result;

result=(a == b)&&(c > b);
printf("(a == b) && (c > b) equals to %d \n", result);

result=(a ==b)&&(c < b);
printf("(a == b) && (c < b) equals to %d \n", result);

result=(a ==b)||(c < b);
printf("(a == b) || (c < b) equals to %d \n", result);

result=(a != b)||(c < b);
printf("(a != b) || (c < b) equals to %d \n", result);

result=!(a != b);
printf("!(a != b) equals to %d \n", result);

result=!(a == b);
printf("!(a == b) equals to %d \n", result);

return 0;
}

```

Output

```

(a == b) && (c > b) equals to 1
(a == b) && (c < b) equals to 0
(a == b) || (c < b) equals to 1
(a != b) || (c < b) equals to 0
!(a != b) equals to 1
!(a == b) equals to 0

```

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &(and), |(or), and ^(ex-or) is as follows –

P	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then –
 A value in binary is : 0011 1100

B Value is binary is : 0000 1101

Operator	Description	Example
&	Binary AND Operator gives 1 if it exists in both operands value As 1.	(A & B) = 12 i.e., 0000 1100
	Binary OR Operator copies a bit 1 if it exists in either operand.	(A B) = 61 i.e., 0011 1101
^	Binary XOR Operator copies the bit 1 if it is set in one operand but not both.	(A ^ B) = 49 i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61 i.e., 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators

The following table lists the assignment operators supported by the C language –

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
=	Bitwise inclusive OR and assignment operator.	C = 2 is same as C = C 2

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Example : Assignment Operators

// C Program to demonstrate the working of assignment operators

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

```
int main()
```

```
{
```

```
int a =5, c;
```

```
    c = a;
```

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

```
    c += a;// c = c+a
```

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

```
    c -= a;// c = c-a
```

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

```
c*= a;// c = c*a
```

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

```
c/= a;// c = c/a
```

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

```
c%= a;// c = c%a
```

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

```
return0;
```

```
}
```

Output

```
c = 5
```

```
c = 10
```

```
c = 5
```

```
c = 25
```

```
c = 5
```

```
c = 0
```

Misc Operators

Besides the operators discussed above, there are a few other important operators including sizeof and ?: (ternary) supported by the C Language.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the address of the variable.
*	Pointer to a variable.	*a gives value at a
? :	Conditional operator.	If Condition is true then value of X or value Y

Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

The sizeof operator

The sizeof is an unary operator which returns the size of data (constant, variables, array, structure etc).

Example :sizeof Operator

```
#include<stdio.h>
int main()
{
int a;
float b;
double c;
char d;
printf("Size of int=%lu bytes\n",sizeof(a));
printf("Size of float=%lu bytes\n",sizeof(b));
printf("Size of double=%lu bytes\n",sizeof(c));
printf("Size of char=%lu byte\n",sizeof(d));
return0;
}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

C Ternary Operator (?:)

A conditional operator is a ternary operator, that is, it works on 3 operands.

Conditional Operator Syntax

```
conditionalExpression ?expression1 : expression2
```

The conditional operator works as follows:

- The first expression conditionalExpression is evaluated at first. This expression evaluates to 1 if it's and evaluates to 0 if it's false.

- If conditionalExpression is true, expression1 is evaluated.
- If conditionalExpression is false, expression2 is evaluated.

Example : C conditional Operator

```
#include<stdio.h>
int main(){
charFebruary;
int days;
printf("If this year is leap year, enter 1. If not enter any integer: ");
scanf("%c",&February);

// If test condition (February == '1') is true, days equal to 29.
// If test condition (February =='1') is false, days equal to 28.
days=(February=='1')?29:28;

printf("Number of days in February = %d",days);
return0;
}
```

Output

If this year is leap year, enter 1. If not enter any integer: 1
 Number of days in February = 29

Other operators such as & (reference operator), * (dereference operator) and -> (member selection) operator will be discussed in C pointers

Operators Precedence and Associativity

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> .	Left to right
Unary	+ - ! ~ ++ -- (type cast)* &sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<<>>	Left to right
Relational	<<=>>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right

Bitwise XOR	\wedge	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

Associativity of an operators

In programming languages, the associativity of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses.

Operators may be associative (meaning the operations can be grouped arbitrarily), left-associative (meaning the operations are grouped from the left), right-associative (meaning the operations are grouped from the right). The associativity and precedence of an operator is a part of the definition of the programming language; different programming languages may have different associativity and precedence for the same type of operator.

Example

Try the following example to understand operator precedence in C –
`#include<stdio.h>`

```
main(){

int a =20;
int b =10;
int c =15;
int d =5;
int e;

    e =(a + b)* c / d;// ( 30 * 15 ) / 5
printf("Value of (a + b) * c / d is : %d\n", e );

    e =((a + b)* c)/ d;// (30 * 15 ) / 5
printf("Value of ((a + b) * c) / d is : %d\n", e );

    e =(a + b)*(c / d);// (30) * (15/5)
printf("Value of (a + b) * (c / d) is : %d\n", e );

    e = a +(b * c)/ d;// 20 + (150/5)
printf("Value of a + (b * c) / d is : %d\n", e );

return0;
```

}

When you compile and execute the above program, it produces the following result

Value of $(a + b) * c / d$ is : 90

Value of $((a + b) * c) / d$ is : 90

Value of $(a + b) * (c / d)$ is : 90

Value of $a + (b * c) / d$ is : 50

Evaluation of expressions

- Operators are symbols which take one or more operands or expressions and perform arithmetic or logical computations.
- Operands are variables or expressions which are used in conjunction with operators to evaluate the expression.
- Combination of operands and operators form an expression.
- Expressions are sequences of operators, operands, and punctuators that specify a computation.
- Evaluation of expressions is based on the operators that the expressions contain and the context in which they are used.
- Expression can result in a value and can produce side effects.
- A side effect is a change in the state of the execution environment.
- An expression is any valid set of literals, variables, operators, operands and expressions that evaluates to a single value.
- This value can be a number or a logical value.
- For instance $a = b + c$; denotes an expression in which there are 3 operands a, b, c and two operator + and =.
- A statement, the smallest independent computational unit, specifies an action to be performed.
- In most cases, statements are executed in sequence.
- The number of operands of an operator is called its arity.
- Based on arity, operators are classified as nullary (no operands), unary (1 operand), binary (2 operands), ternary (3 operands).

Type Conversion

When variables and constants of different types are combined in an expression then they are converted to same data type. The process of converting one predefined type into another is called type conversion. Type conversion in c can be classified into the following two types:

Implicit Type Conversion

When the type conversion is performed automatically by the compiler without programmers intervention, such type of conversion is known as implicit type conversion or type promotion. The compiler converts all operands into the data type of the largest operand.

It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it.

Also, conversion of float to int causes truncation of fractional part, conversion of double to float causes rounding of digits and the conversion of long int to int causes dropping of excess higher order bits.

Explicit Type Conversion

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion. The explicit type conversion is also known as type casting.

Type casting in c is done in the following form:

`(data_type)expression;`

where, data_type is any valid c data type, and expression may be constant, variable or expression.

For example,

`x=(int)a+b*d;`

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

All integer types to be converted to float.

All float types to be converted to double.

All character types to be converted to integer.

Solved Examples

1. The number of tokens in the following C statement.

```
printf("i = %d, &i = %x", i, &i);
```

(a) 3

(b) 26

(c) 10

(d) 21

Answer (c)

Explanation: In a C source program, the basic element recognized by the compiler is the "token." A token is source-program text that the compiler does not break down into component elements. There are 6 types of C tokens : identifiers, keywords, constants, operators, string literals and other separators. There are total 10 tokens in the above printf statement.

2. What is the output of the following

```
#include "stdio.h"
```

```
int main()
```

```
{
```

```
int x, y = 5, z = 5;
```

```
    x = y == z;
```

```
printf("%d", x);
```

```
getchar();
```

```
return 0;
```

```
}
```

Output : 1

Explanation: The crux of the question lies in the statement `x = y==z`. The operator `==` is executed before `=` because precedence of comparison operators (`<=`, `>=` and `==`) is higher than assignment operator `=`. The result of a comparison operator is either 0 or 1 based on the comparison result. Since y is equal to z, value of the expression `y == z` becomes 1 and the value is assigned to x via the assignment operator.

3.

```
#include <stdio.h>
int main()
{
    inti = 3;
    printf("%d", (++i)++);
    return 0;
}
```

What is the output of the above program?

- (A) 3
- (B) 4
- (C) 5
- (D) Compile-time error

Answer: (D)

Explanation: In C, prefix and postfix operators need l-value to perform operation and return r-value. The expression `(++i)++` when executed increments the value of variable `i` (`i` is a l-value) and returns r-value. The compiler generates the error(l-value required) when it tries to post-increment the value of a r-value.

4. What is the output of following program?

```
#include <stdio.h>
int main()
{
    int a = 1;
    int b = 1;
    int c = a || --b;
    int d = a-- && --b;
    printf("a = %d, b = %d, c = %d, d = %d", a, b, c, d);
    return 0;
}
```

- (A) a = 0, b = 1, c = 1, d = 0
- (B) a = 0, b = 0, c = 1, d = 0
- (C) a = 1, b = 1, c = 1, d = 1
- (D) a = 0, b = 0, c = 0, d = 0

Answer: (B)

Explanation: Let us understand the execution line by line.

Initial values of `a` and `b` are 1.

```
// Since a is 1, the expression --b is not executed because
// of the short-circuit property of logical or operator
// So c becomes 1, a and b remain 1
```

```
int c = a || --b;
```

```
// The post decrement operator -- returns the old value in current expression
// and then updates the value. So the value of expression --a is 1. Since the
// first operand of logical and is 1, shortcircuiting doesn't happen here. So
// the expression --b is executed and --b returns 0 because it is pre-increment.
// The values of a and b become 0, and the value of d also becomes 0.
```

```
int d = a-- && --b;
```

5. What is the output

```
#include<stdio.h>
int main(void)
{
int a = 1;
int b = 0;
    b = a++ + a++;
printf("%d %d",a,b);
return 0;
}
```

The output of program is also undefined. It may be 3, 4, or may be something else. The subexpression `i++` causes a side effect, it modifies `a`'s value, which leads to undefined behavior since `a` is also referenced elsewhere in the same expression.

6. What is the output of the following

```
#include <stdio.h>
int main()
{
printf("%d", 1 << 2 + 3 << 4);
return 0;
}
```

- (A) 112
- (B) 52
- (C) 512
- (D) 0

Answer: (C)

Explanation: The main logic behind the program is the precedence and associativity of the operators. The addition (+) operator has higher precedence than shift (<<) operator. So, the expression boils down to $1 \ll (2 + 3) \ll 4$ which in turn reduces to $(1 \ll 5) \ll 4$ as the shift operator has left-to-right associativity.

7. What is the output of the following

```
#include<stdio.h>
int main()
{
int a = 2,b = 5;
    a = a^b;
    b = b^a;
printf("%d %d",a,b);
return 0;
}
```

- (A) 5 2
- (B) 2 5
- (C) 7 7
- (D) 7 2

Answer: (D)

Explanation: ^ is bitwise xor operator.

$a = 2$ (10) $b = 5$ (101)

$a = a \wedge b$ (10 ^ 101) = 7(111)

$b = a \wedge b$ (111 ^ 101) = 2(10)

8. What is the output of the following

```
# include <stdio.h>
int main()
```

```
{
int x = 10;
int y = 20;
    x += y += 10;
printf (" %d %d", x, y);
return 0;
}
```

(A) 40 20(B) 40 30(C) 30 30(D) 30 40

Answer: (B)

Explanation: The main statement in question is “x += y += 10”. Since there are two += operators in the statement, associativity comes into the picture. Associativity of compound assignment operators is right to left, so the expression is evaluated as x += (y += 10).

Questions

- 1) Describe two major components of a computer.
- 2) What is the purpose of operating system.
- 3) Describe two major categories of software.
- 4) Write a short essay on Algorithm, Flowchart, Pseudo-code quoting a suitable example.
- 5) Describe the basic steps in system development life cycle.
- 6) Draw a flowchart to find biggest number in the given list of 4 integers.
- 7) Write a program that reads nine integers and prints them three in a separated by commas .
- 8) Write an algorithm and flowchart to find sum of given n numbers.
- 9) Explain bitwise operators with examples.
- 10) Write a program that extracts and prints the second right most digit of the integral portion of a float.
- 11) A Fibonacci number is a member of a set in which each number is the sum of the previous two numbers. Write a program to print 6 Fibonacci numbers, you are to use only three variable fib1, fib2, fib3.
- 12) What is the result of the following expressions if a is a 8 bit unsigned integer a) a & ~a b) a|~a c) a^~a d) a & 0xFF
- 13) Write the name of editor and compiler you use in lab for c programming and give commands to invoke them.
- 14) Evaluate the expression a) ((5<=7) && (6 <3)) b) !(9>5)|| (2<5)

<< Always good to read good Text Books >>

Unit – II

Selection and Iterative constructs : *if, if-else, nested if, else-if ladder, Block of statements and scope. switch and break statements. Examples – student grade decision based on marks obtained, simple calculator program.*

Loop statements : *Syntax and behavior of for, while and do-while looping constructs.*

Other statements related to looping – break, continue, goto programming examples – finding the gcd of two numbers, finding the factorial of the given number, generating prime numbers etc.

Selection and Iterative constructs

Any Program could be written with three constructs **sequence, selection and loop**. Sequence constructs are those that execute all the statements from top to bottom one time, **selection** constructs allows to choose between two or more alternative group of statements based on decision making and loop constructs executes group of statements zero, one or more times based on decision.

Fortunately our world is filled with choices one example is whether i) to have tea or not ii) to have tea or coffee iii) to have tea or coffee or milk or soft drink or water or nothing iv) if tea is selected among tea and coffee then normal tea or lemon tea if normal tea is selected then if biscuits or cookies, if coffee is selected then cold coffee or normal coffee etc.

1. Example (i) given above

```
if (want tea)
    serve tea
```

2. Example (ii) given above

```
if (want tea)
    serve tea
else
    serve coffee
```

3. Example (iii) given above

```
if (want tea)
    serve tea
else
    if (want coffee)
        serve coffee
    else
        if (want milk)
```

```

        serve milk
    else
        if(want soft drink)
            serve soft drink
        else
            if(want water)
                serve water
            else
                serve nothing
    
```

4. Example(iv)given above

```

    {
        if (want tea)
            {
                if(want normal tea)
                    {
                        if(want biscuits)
                            serve normal tea with biscuits
                        else
                            serve normal tea with cookies
                    }
                else
                    serve lemon tea
            }
        else
            {
                if(want cold coffee)
                    serve cold coffee
                else
                    serve normal coffee
            }
    }
    
```

Data are called **logical** if they convey the idea of yes or no, in computer science we use **true** or **false**. In c if a data value is zero it is considered as false, if it is a non zero either positive or negative value considered as true.

To write logical expressions generally we use comparative operators which include relative and equality operators.

Relational operators : < ,> ,<=, >=

Equality operators : == , !=

A logical expression may be a **simple condition** contains only one relational operator, or a **compound condition** which contains more than one simple condition combined with logical operators. Final value of any logical expression is true or false i.e. **either 1 or 0**. Generally we represent 1 for true and 0 for false.

Simple logical expressions : $a < b$, $c == d$, $5 > 9$, $num1 \leq num2$ etc.

Compound conditions : $(a < b) \&\& (x > y)$, $(num1 != 5) \|\| (rem == 0) \&\& (a > 5)$ etc.

Program 1.

```

// C Program to demonstrate the working of relational operators
#include<stdio.h>
int main()
{
    
```

```
int a =5, b =5;
```

```
printf("%d == %d = %d \n", a, b, a == b);// true
printf("%d > %d = %d \n", a, b, a > b);//false
printf("%d < %d = %d \n", a, b, a < b);//false
printf("%d != %d = %d \n", a, b, a != b);//false
printf("%d >= %d = %d \n", a, b, a >= b);//true
printf("%d <= %d = %d \n", a, b, a <= b);//true
return 0;
}
```

Guess how many values will you get 1 and how many values will you get 0 : -

in the above program a value is 5, b value is also 5, so $a==b$, $a<=b$ and $a>=$ are true and its value is 1, remaining $a!=b$, $a>b$ and $a<b$ are false its value is 0.

Output

```
5 == 5 = 1
5 > 5 = 0
5 < 5 = 0
5 != 5 = 0
5 >= 5 = 1
5 <= 5 = 1
```

Program 2.

// C Program to demonstrate the working of logical operators

```
#include<stdio.h>
int main()
{
int a =5, b =5, c =10, result;

result=(a = b)&&(c > b);
printf("(a = b) && (c > b) equals to %d \n", result);
result=(a = b)&&(c < b);
printf("(a = b) && (c < b) equals to %d \n", result);

result=(a = b)||(c < b);
printf("(a = b) || (c < b) equals to %d \n", result);

result=(a != b)||(c < b);
printf("(a != b) || (c < b) equals to %d \n", result);

result=!(a != b);
printf("!(a != b) equals to %d \n", result);

result=!(a == b);
printf("!(a == b) equals to %d \n", result);

return 0;
}
```

In the above program we are using logical and (&&) , logical or(||) and logical not(!).

Output

(a = b) && (c > b) equals to 1
 (a = b) && (c < b) equals to 0
 (a = b) || (c < b) equals to 1
 (a != b) || (c < b) equals to 0
 !(a != b) equals to 1
 !(a == b) equals to 0

Selection Constructs

Decision-making structures (Selection constructs) require that the programmer specifies one or more conditions to be evaluated by the program, along with statements to be executed if the condition is determined to be true, other statements to be executed if the condition is determined to be false.

C programming language provides the following types of decision-making statements.

if statement , an if statement consists of a expression(condition) followed by one or more statements.

if...else statement an if statement executes if expression is true can followed by an else statement, executes if expression is false.

nested if statements You can use one if or else if statement inside another if or else if statement(s)

If – else ladder among the many options if it is required to select one then it is used.

switch statement a switch statement allows a variable to be tested for equality against a list of values.

if Statement

An **if** statement consists of a Boolean expression followed by one or more statements. A set of statements after ‘if’ is called if block. For this block of statements multiple statements can be combined into a compound statement through the use of braces, if only one statement is there under if no need to include in braces, it is better always practice using braces. No semicolon is needed for an if statement, but statements under if required semi colon.

Syntax

The syntax of an ‘if’ statement is:

```
if(expression)
{
    /* statement(s) will execute if the expression is true its value is 1 */
}
```


If the expression evaluates is **true**, then the if-block will be executed. If the expression evaluates to **false**, then the first line after the end of the ‘if’ statement (after the closing curly brace) will be executed.

Program .

```
// C Program to demonstrate the working of simple if
#include <stdio.h>

int main ()

{
    /* local variable definition */
    int a;
    scanf("%d",&a);
    /* check the Boolean condition using if statement */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        a=20;
    }
    printf("value of a is : %d\n", a);
    return 0;
}
```

Explanation : if input value is less than 20 the value will be changed to 20, otherwise the value will be as it is.

if...else Statement

An **if** statement can be followed by an **else** statement, if expression value is true ‘if’ statements will executes, if expression value is false ‘else’ statements will executes. A set of statements after ‘if’ is called if block. A set of statements after else is called else block.

For this blocks of statements multiple statements can be combined into a compound statement through the use of braces, if only single statement is there no need to include in braces, it is better always practice using braces. No semicolon is needed for an ‘if’ and ‘else’ statements, but statements under if and else required semi colon.

Syntax

The syntax of an **if...else** statement is:

```
if(expression)
{
    /* statement(s) will execute if the expression is true, value is 1*/
}
else
{
    /* statement(s) will execute if the expression is false , value is 0*/
}
```

Example for if-else

Program to calculate discount amount on the given amount, if amount less than 1000 discount rate is 5%, otherwise discount rate is 10%

```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a;
    float dis;
    scanf("%d",&a);
    /* check the boolean condition */
    if( a <1000 )
    {
        /* if condition is true then discount rate is 5% */
        dis=a*5.0/100;
    }
    else
    {
        /* if condition is false when amount is 1000 or more than 1000 then
        discount rate is 10% */
        dis=a*10.0/100;
    }
    //printing the calculated discount
    printf("Discount is : %f\n", dis);
    return 0;
}
```

Nested if Statements

For if-else the statements may be any valid c statements, including another if-else. When an if-else is included within an if-else it is known as a nested if statement.

It is always legal in C programming to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s). Else is always paired with the most recent unpaired if.

Syntax

The syntax for a **nested if** statement is as follows:

```
if( expression 1)
{
    /* Executes when the expression 1 is true */
    if(expression 2)
    {
        /* Executes when the expression 2 is true */
    }
}
```

```

}
else
{
    /* Executes when the expression 1 is false */
    if(expression 3)
    {
        /* Executes when the expression 3 is true */
    }
}
}

```

Program 5.

Write a program to find biggest among the given three numbers.

```

//program to demonstrate nested - if
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a,b,c;
    /*accepting the values for a,b,c */
    printf("\n Enter three integers :");
    scanf("%d%d%d",&a,&b,&c);
    /* check the boolean condition for a and b*/
    if( a > b )
    {
        /* if condition is true then check for a and c */
        if( a > c )
        {
            /* if condition is true then print a */
            printf("\n A value is biggest" );
        }//inner if
        else
        {
            /* if condition is false then print c*/
            printf("\n C value is biggest" );
        }
    }//outer if
    else
    {
        /* if condition is false then check for b and c */
        if( b > c )
        {
            /* if condition is true then print b */
            printf("\n B value is biggest" );
        }//inner if
        else
        {
            /* if condition is false then print c*/
            printf("\n C value is biggest" );
        }//inner else
    }
}

```

```

    }//outer else
    return 0;
} //main

```

if...else if...else Statement – if else ladder

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement. This is also one type of nested if, but generally nesting happens in 'else' not in 'if'.

Syntax

The syntax of an **if...else if...else** statement in C programming language is:

```

if(expression 1)
{
    /* Executes when the expression 1 is true */
}
    else if( expression 2)
    {
        /* Executes when the expression 2 is true */
    }
        else if( expression 3)
        {
            /* Executes when the expression 3 is true */
        }
            else
            {
                /* executes when the none of the above condition is
true */
            }
}

```

Example

Write a c program to calculate discount amount if choice is 1 discount rate is 5%, if choice is 2 discount rate is 10%, if choice is 3 discount rate is 15% for any other option discount is 0

```

#include <stdio.h>
int main ()
{
    /* local variable definition */
    int amt,ch;
    float dis;
    /* accept the input values*/
    scanf("%d%d",&amt,&ch);

    if(ch == 1 )
    {
        /* if condition is true dis is 5% */
        dis=amt*5.0/100;
    }
}

```

```

else if( ch == 2 )
{
    /* if condition is true dis is 10% */
    dis=amt*10.0/100;
}
else if(ch == 3 )
{
    /* if condition is true dis is 15% */
    dis=amt*15.0/100;
}
else
{
    /* if none of the conditions is true)*/
    dis=0;
}

printf("Discount is : %f\n", dis );
return 0;
} //end of main
    
```

switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax

The syntax for a **switch** statement in C programming language is as follows:

```

switch(expression)
{
    case constant-expression :
        statement(s);
        break; /* optional */
case constant-expression :
    statement(s);
    break; /* optional */
/* you can have any number of case statements */
default : /* Optional */
    statement(s);
}
    
```

The following rules apply to a **switch** statement:

The **expression** used in a **switch** statement must have an integer or character. You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon. The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a variable.

When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached. When a **break** statement is reached, the switch

terminates, and the flow of control jumps to the next line following the switch statement.

Not every case needs to contain a **break**. If no **break** appears, the flow of control will continue to subsequent cases until a break is reached.

A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Program

Write a c program to accept a number called num1, if the number is 9 square num1, if number is 10 accept another number to num1 or if number is 2 or 3 add 100 to num1, otherwise make the num1 to 0 and display the num1.

Example

```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int num1;
    /* input the num1 */
    printf("\n Enter a integer :");
    scanf("%d",&num1);
    switch(num1)
    {
        case 9 :
            num1*=num1;
            break;
        case 10 :
            /* accept the new number to num1*/
            printf("\n Enter a integer :");
            scanf("%d",&num1);
            break;
        case 2 :
        case 3 :
            /*if option is 2 or 3 add 100 to num1*/
            num1=num1+100;
            break;
        default :
            num1=0;
    }
    /* display the output*/
    printf("\nFinal number is %d\n", num1 );
    return 0;
}
```

Program 9.

Write a program to accept the three subject marks, display the message fail if in any one

subject marks are less than 50, other wise calculate the average and print the class if it is more than 74% result is distinction, if 51%-59% display second otherwise display first.

Explanation :

Here the task is to accept 3 subject marks and display the result, inputs are 3 integer numbers and output is either “fail” or one among “second”, “first” and “distinction”, the variables we required are 3 integers and one float type variable for storing average marks.

We can write this program using logical operators with compound expressions or with simple expressions, we write this program in both the ways

Program using compound expressions :

```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int num1;
    /* accept the marks for three subjects*/
    printf("\n Enter 3 subjects marks : ");
    scanf("%d%d%d",&m1,&m2,&m3);
    if( m1<50||m2<50||m3<50 )
    {
        /* if one condition is true result is true */
        printf("\Fail");
    }
    else
    {
        avg=(m1+m2+m3)/3.00;
        if(avg<60)
            printf("\n second");
        else
            if(avg<75)
                printf("\n First");
            else
                printf("\n Distinction");
    }
    return 0;
} //end of main
```

Program

Write a c program to accept a character and if entered character is uppercase convert it into lowercase and if entered character is lowercase convert it to uppercase and display both the characters.

Task required one character variable to accept the character and one variable required to store converted character.

```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    char ic,oc;
    /* accept the character*/
    printf("\n Enter one character : ");
    scanf("%c",&ic);

    if(ic>'A' &&ic<'Z')
    {
        oc=ic+32; // arithmetic operation is possible on char variable
        printf("\n Entered character is %c, converted is %c ",ic,oc);
    }
    else
    if(ic>'a' &&ic<'z')
    {
        oc=ic-32; arithmetic operation is possible on char variable
        printf("\n Entered character is %c, converted is %c ",ic,oc);
    }
    else
        printf("\n you have entered wrong character ");
}
```


Questions

I . What is output of the following codes and give explanations.

- a) if originally x=0,y=0 and z=1 what is value of x,y and z after executing following code

```
if (x)
if (y)
    z=3;
else
    z=2;
```

- b) if originally x=4,y=0 and z=2 what is value of x,y and z after executing following code

```
if (z= =0 || x && !y)
if (!z)
    y=1;
else
    x=2;
```

- c) if originally x=0,y=0 and z=1 what is value of x,y and z after executing following code

```
if (z= y)
{
    y++;
    z--;
}
else
    --x;
```

- d) if originally x=0,y=0 and z=1 what is value of x,y and z after executing following code

```
switch(x)
{
case 0 : x=2;
        y=3;
case 1 :
        x=4;
default :
        y=3;
        x=1;
}
```

- e) #include <stdio.h>

```
void main()
{
int x = 5;
```

```

if (x < 1)
printf("hello");
if (x == 5)
printf("hi");
else
printf("no");
}
f.
#include <stdio.h>
void main()
{
int x =0;
if(x ==0)
printf("hi");
else
printf("how are u");
printf("hello");
}
    
```

1. Write a C program to find maximum between two numbers.
2. Write a C program to check whether a number is even or odd.
3. Write a C program to check whether a year is leap year or not.
4. Write a C program to check whether a number is negative, positive or zero.
5. Write a C program to check whether a number is divisible by 5 and 11 or not.
6. Write a C program to input any alphabet and check whether it is vowel or consonant.
7. Write a C program to input any character and check whether it is alphabet, digit or special character.
8. Write a C program to input angles of a triangle and check whether triangle is valid or not.
9. Write a C program to find all roots of a quadratic equation.
10. Write a C program to input electricity unit charges and calculate total electricity bill according to the given condition:
 For first 50 units Rs. 0.50/unit
 For next 100 units Rs. 0.75/unit
 For next 100 units Rs. 1.20/unit
 For unit above 250 Rs. 1.50/unit
 An additional surcharge of 20% is added to the bill
11. Write a C program to input marks of five subjects Physics, Chemistry, Biology, Mathematics and Computer. Calculate percentage and grade according to following:
 Percentage > 90% : Grade A
 Percentage > 80% : Grade B
 Percentage > 70% : Grade C
 Percentage > 60% : Grade D
 Percentage > 40% : Grade E
 Percentage < 40% : Grade F
12. Write a C program to input basic salary of an employee and calculate its Gross salary according to following:
 Basic Salary >= 10000 : HRA = 20%, DA = 80%
 Basic Salary >= 20000 : HRA = 25%, DA = 90%
 Basic Salary >= 30000 : HRA = 30%, DA = 95%

Looping statements

Program 1.

```
#include <stdio.h>
int main ()

{
printf(" Welcome to CVR College of Engineering ");
printf(" Welcome to CVR College of Engineering ");
printf(" Welcome to CVR College of Engineering ");
printf(" Welcome to CVR College of Engineering ");
printf(" Welcome to CVR College of Engineering ");
}
```

Above program prints the message " Welcome to CVR College of Engineering " 5 times. The program contains 5 printf statements. In c it is possible that we write only one printf statement in the program but the program prints the message 5 times.

Program 2.

```
#include <stdio.h>
int main ()
{
inti=0;

i=i+1;
printf(" \n The Value is %d ",i);

i=i+1;
printf(" \n The Value is %d ",i);

i=i+1;
printf(" \n The Value is %d ",i);

i=i+1;
printf(" \n The Value is %d ",i);

i=i+1;
printf(" \n The Value is %d ",i);
}
```

The above program prints the output as

```
The Value is 1
The Value is 2
The Value is 3
The Value is 4
The Value is 5
```

In the above program the two statements

```
i=i+1;
printf(" \n The Value is %d ",i);
```

repeated five times. The statements are looking same. The statements are performing same operations i.e. i)incrementing ii)printing five times. Instead of typing above two statements 5

times we will type only once in the program and we use looping constructs to repeat the same statements five times. You may encounter situations when a block of code needs to be executed several number of times.

The real power of computers is in their ability to repeat an operation many times. This repetition is called looping.

Program

```
#include <stdio.h>
int main ()

{
while(1)
{
printf(" Welcome to CVR College of Engineering ");
}
}
```

in the above program one printf statement is written under while. The behaviour of while is if condition value is true it repeats the statements under while. The condition value is always true in the above program, so it prints the message “ Welcome to CVR College of Engineering “ infinitely.

Program

```
#include <stdio.h>
int main ()
{
while(0)
{
printf(" Welcome to CVR College of Engineering ");
}
}
```

in the above program one printf statement is written under while. According to the behaviour of while if the condition value is true it repeats the statements under while, if condition value is false it does not execute the statements under while, so the above program does not display any output.

When ever we are writing any looping constructs in a program we must write a condition such that it is true for some repetitions and condition should becomes false then the loop stops repeating the statements.

A loop statement allows us to execute a statement or group of statements multiple times.

Pre test and Post test loops

The condition we write along with loop will be true for some iterations then it should become false. In the above program 4 and program 5 directly we have written 1 and 0 in place of condition. The condition we write is to control the loop is called loop control expression.

Pre testloop :

In each iteration, the control expression is tested first, if it is true the loop continues, otherwise loop terminates

Post test loop

In each iteration the loop statements executes first then the control expression is tested. If it is true a new iteration is started otherwise the loop terminates.

In any looping constructs three processes takes place

1. Initialization
2. Checking loop control expression

3. Loop control variable update.

Below table compares the above processes among the pre test and post test.

Pre test loop	Post test
Initialization	1
Number of comparisons	n+1n
Updation n n	
Minimum iterations	01

The loops can be summarised in two categories 1. Event controlled 2. Counter controlled.

Event controlled : An event changes the control expression from true to false. For example find sum of integers entered through the keyboard until entered number is 9.

Counter controlled : Counter controlled loop is a form in which the number of iterations to be performed is known in advance. For example find the sum of 10 integers entered through the keyboard.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages:

C programming language provides the following types of loops to handle looping requirements.

while loop (it is Pre test loop and event controlled loop) : Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

for loop (it is Pre test loop and counter controlled loop) Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

do...while loop(it is Post test loop and event controlled loop) It is more like a while statement, except that it tests the condition at the end of the loop body.

while Loop

A **while** loop in C programming repeatedly executes a set of statements or statement as long as a given condition is true. while statement consists of while followed by controlling expression enclosed within parentheses.

Syntax

The syntax of a **while** loop in C programming language is:

```
while(condition)
{
    statement(s);
}
```

Here statement(s) may be a single statement or a block of statements. The **condition** may be any expression, and true for any nonzero value. The loop iterates while the condition is true. After executing the while body the program control returns back to the while expression. When

the condition becomes false, the program control passes to the line immediately following the loop.

Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

While uses a condition along with it, based on condition value the statements will be repeated, as we know any condition value would be 1 or 0.

program 5.

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 1; // 1. initialization
    /* while loop execution */
    while( a < 20 ) // condition is true until 'a' value is less than 20
        // 2. loop control expression is 'a < 20'
    {
        printf("value of a: %d\n", a);
        a=a+1 // 3. loop control variable updatation
    }
    return 0;
}
```

for Loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. for consists three expressions separated by semicolons and enclosed within parentheses.

Syntax

The syntax of a **for** loop in C programming language is:

```
for ( expression 1; expression 2; expression 3 )
{
    statement(s);
}
```

Generally expression 1 is initialization
 expression 2 is loop control or condition
 expression 3 is increment or decrement

Here is the flow of control in a 'for' loop:

1. The **initialization** step is executed first, and only once. This step allows you to initialize any loop control variables.
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed.
3. After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates, control jumps to the next statement just after the 'for' loop.

Program.

// Above program 5 with for

Example

```
#include <stdio.h>
int main ()

{
    /* for loop execution, all the three processes come in for at one place */
    for(int a = 1; a < 20; a = a + 1 )
    {
        printf("value of a: %d\n", a);
    }
    return 0;
}
```

do...while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

The do-while statement consists of a do keyword followed by statement or statements that constitute do-while body, followed by the while clause consisting of while keyword followed by do-while controlling expression enclosed with in parenthesis. The while clause is terminated with a semicolon.

Syntax

The syntax of a **do...while** loop in C programming language is:

```
do

{
    statement(s);
```

```
}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

If condition value is false the program control is transferred to the statement present next to the do-while statement.

The body of loop executes once even if controlling expression is initially false.

Program .

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 1; // 1.initialization
    /* do loop execution */
    do
    {
        printf("value of a: %d\n", a);
        a = a + 1; // 3.updation of loop control variable
    }while( a < 20 );//2. Loop control condition
    return 0;
}
```

Program.

Write a program to find the sum of numbers entered through the keyboard until the number is 9.

This program comes under event controlled category we can use while or do-while.

```
int main ()
{
    /* local variable definition */
    int a, sum=0
    scanf("%d",&a); // accepting the first number
    while( a !=9 ) // condition to check the number is 9 or not
    { // beginning of while
        sum=sum+a; // finding the sum of entered numbers
        scanf("%d",&a);//accepting 2nd number, 3rd number until number is 9
    }//ending of while
    printf("\n The sum of entered numbers is %d: ",sum);
    return 0;
}
```


Program

Write a program to print the table of a given number.

This program comes under counter controlled, we use for

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int i,p,n; // 'i' is for loop control, 'p' is for product, 'n' is input

    for(i = 1; i<= 10; i = i + 1 )
    {
        p= n*i;
        printf("\n %d * %d = %d", n,i,p);

    }
    return 0;
}
```

Loop Control Statements

Loop control statements change execution from its normal sequence.

C supports the following loop control statements.

break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.

continue statement Causes the loop to skip (by passes) the remainder of its body and immediately retest its condition prior to next iteration.

goto statement Transfers control to the labelled statement.

break Statement

The **break** statement in C programming has the following two usages:

When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. It can be used to terminate a case in the **switch** statement .

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in C is as follows:

```
break;
```

Program 10.

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;
    /* while loop execution */
    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
        if( a > 15)
        {
            /* terminate the loop using break statement */
            break;
        }
    }
    return 0;
}
```

Program 11.

Program to find a number is prime or not.

```
int main ()
{
    /* local variable definition */
    inti, j=2;
    scanf("%d",&i);

    while(j<i)
    {
        r=i%j;
        if(r==0)
        {
            printf("\n Number is not a prime");
            break;
        }
        j++;
    }

    // checking how the control came out from the loop
    if(j ==i) //checking whether condition became false
    printf("%d is prime\n", i);
}
```

```

        return 0;
    }

```

continue Statement

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination of loop, it bypasses the current iteration of the loop to take place, skipping any code in between.

For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

Syntax

The syntax for a **continue** statement in C is as follows:

```
continue;
```

Program

```

#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a = 10;
    /* do loop execution*/
    do
    {
        if( a == 15)
        {
            /* skip the iteration */ a = a + 1;
            continue;
        }
        printf("value of a: %d\n", a);
        a++;
    }while( a < 20 );
    return 0;
}

```

When the above code is compiled and executed, it produces the following result:

```

value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 16 value
of a: 17 value of a: 18 value of a: 19

```

goto Statement

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labelled statement in the same function.

Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

Syntax

The syntax for a **goto** statement in C is as follows:

```
goto label;

..

.

label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below the **goto** statement.

Program .

```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a = 10;
    /* do loop execution */
    LOOP:do
    {
        if( a == 15)
        {
            /* skip the iteration */
            a = a + 1;
            goto LOOP;
        }
        printf("value of a: %d\n", a);
        a++;
    }while( a < 20 );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 16 value
of a: 17 value of a: 18 value of a: 19
```

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions make an endless loop.

Program .

```
#include <stdio.h>

int main ()
{
```

```

    for( ; ; )
    {
        printf("This loop will run forever.\n");
    }
    return 0;
}
    
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the `for(;;)` construct to signify an infinite loop.

You can terminate an infinite loop by pressing `Ctrl + C` keys.

Nested Loops

We write statement or statements under a loop, that statement could be once again a loop. C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept. There may be `for` inside a `while` or `while` inside a `for` or `for` inside a `for` etc.

Syntax

The syntax for a **nested for loop** statement in C is as follows:

```

for ( initialization; condition; increment )
{
    for ( initialization; condition; increment )
    {
        statement(s);
    }
    statement(s);
}
    
```

The syntax for a **nested while loop** statement in C programming language is as follows:

```

while(condition)
{
    while(condition)
    {
        statement(s);
    }
    statement(s);
}
    
```

The syntax for a **nested do...while loop** statement in C programming language is as follows:

```

do
{
    statement(s);
do
{
    statement(s);
}while( condition );
}while( condition );
    
```

A nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Program

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
#include <stdio.h>
int main ()
{
    /* local variable definition */ inti, j;
    for(i=2; i<100; i++)
    {
        for(j=2; j <= (i/j); j++)
            if(!(i%j)) break;           // if factor found, not prime
        if(j > (i/j)) printf("%d is prime\n", i);
    }

    return 0;
}
```

Program.

Program to print tables from 2 to n

```
int main()
{
    inti,j,n
    scanf ("%d",&n)
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=10;j++)
        {
            printf("\n %d*%d=%d",i,j,i*j);
        }
    }
}
```

Program

Program to find sum of digits of a given number.

```
#include <stdio.h>
int main()
{
    int n, t, sum = 0, remainder;
    printf("Enter an integer\n");
    scanf("%d", &n);
    t = n;
    while (n != 0) // checking number becomes zero
    {
        remainder = n % 10; // extracting left most digit
```

```

sum = sum + remainder; // adding left most digit
n = n / 10; // removing the leftmost digit
    }
    printf("Sum of digits of %d = %d\n", t, sum);
    return 0;
}
    
```

Program .

Program to reverse the number

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

```

int main()
{
int n, reverse = 0;

printf("Enter a number to reverse\n");
scanf("%d", &n);

while (n != 0)
    {
reverse = reverse * 10;
reverse = reverse + n%10;
n = n/10;
    }

printf("Reverse of entered number is = %d\n", reverse);

return 0;
}
    
```

Program.

Write a program to find a number is palindrome or not.

Program

Write a program to find gcd of two numbers

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

```

int main()
{
int a, b, num1, num2, temp, gcd;
printf("Enter First Integer:\t");
scanf("%d", &num1);
printf("\nEnter Second Integer:\t");
scanf("%d", &num2);
    a = num1;
    b = num2;
    
```

```

while (b != 0)
{
temp = b;
    b = a%b;
    a = temp;
}
gcd = a;
printf("\nGreatest Common Divisor of %d and %d = %d\n", num1, num2, gcd);
printf("\n");
return 0;
}
    
```

Program .

Write a program to print n Fibonacci numbers.

Program .

Write a program to find factorial of a number.

Program.

Write a program to check the number is perfect or not.

(An integer is said to be perfect if its factors sum is the number, Ex : 28 its factors are 1,2,4,7,14 its sum is $1+2+4+7+14 = 28$. Ex : 6,28,496)

Program .

Write a program to check a given number is Armstrong or not.

(An Armstrong number is $153=1^3+5^3+3^3$)

Program .

Write a program to Find the sum of series $1^2+2^2+3^2+\dots n$ terms

1. What would be printed from each of the following program segment

- a)

```
x=12;
while(x>7)
{
printf("%d\n",x);
x--;
}
```
- b)

```
for(x=12;x>12;x--)
printf("%d\n",x);
```
- c)

```
x=12;
do
{
printf("%d\n",x);
x--;} while(x>7);
```


Questions

1. Differentiate between continue and break statements.
2. What is pre test and post test loops.
3. Write the differenc between while and do-while loop.
4. Write a program to generate n Fibonacci numbers.
5. Write the program to generate factorial of the integers from 1 to 5.
6. Write program to find GCD of two numbers.
7. Write a c program to print 5 to 1 numbers using for,while and do-while.
8. Write a program to find sum of digits of a number.
9. Trace the following code when expr is replaced with break and continue.


```
for( count=1;count<=10;count++)
{
if(count==6)
expr;
printf("\n count = %d",count);
}
```
10. Develop a flowchart and write a c program to display all prime numbers less than number entered by the user.
11. Define terinary operator with an example.
12. Write a c program to find LCM of given two numbers.
13. Write a c program to find a number is Armstrong or not.
14. Write a c program to find sum of even and odd numbers in the range from 2 to n.

Unit - III

Introduction to structured programming :Preprocessing phase, Preprocessing directives, symbolic constants, macros, conditional compilation, standard symbolic constants

Function Basics : Function header, function prototype specification, function definition and function call. Actual, formal parameters and return statement. Examples – standard library functions and simple user defined functions. Scope and extent of variables.

Storage classes – automatic, register, static and global.

Preprocessing Phase

Before a c program compiled in a compiler source code is processed by a program called pre-processor, this process is called preprocessing.

Commands used in preprocessing are called pre-processor directives, and they begin with # symbol.

1. Macro defines a constant value. Example is #define pi 3.14. cpreprocessor is a text substitution tool and it instructs the compiler to do required preprocessing before the actual compilation.
2. Header file inclusion, #include<filename>. The source code of the filename is included in the main program at the specified place.
3. Conditional compilation , set of commands like #ifdef, #endif, #if, #else, #ifndef are included or excluded in source program before compilation with respect to the condition.

Process of Preprocessing :

Source code →Preprocessor→Expanded source code →compiler

Program 1.

```
#include<stdio.h>
#define height 100
#define number 10.31
#define letter 'A'
main()

{
printf("Values are %d, %f,%c",height,number,letter);
}
```

Program 2.

Parameterized macros

```
#include<stdio.h>
#define square(x) ((x)*(x))

main()
{
int a=3;
printf("Values are %d",square(a));
}
```

Conditional compilation

```
#ifdef syntax
#ifdef sum
    #define sum 10
#endif
```

```
#undef syntax
#ifdef sum
    #undef sum
    #define sum 20
#endif
```

if clause statement is included in source file if given condition is true, otherwise else clause statement is included.

program 3.

```
#include<stdio.h>
#define a 100
main()
{
    #if (a==10)
    printf("if This line will be included");
    #else
    printf("else This line will be included");
    #endif
}
```

Program 4.

```
#include<stdio.h>
#define height 10
main()
{
    printf(" The value of height is %d",height);
}
```

```
#undef height
#define height 20
printf(" The value of height is %d",height);
}
```

Program 5.

```
#include<stdio.h>
#define num 10
main()

{
#ifdef num
printf(" Num is defined , this line will be included");
#else
printf(" Num is not defined");
#endif
}
```

Program 6.

```
#include<stdio.h>
#define num 10
main()

{
#ifdef temp
printf(" temp is not defined , this line will not be included");
#else
printf(" Num is defined");
#endif
}
```

Program 7.

```
#include<stdio.h>
#define sum(a,b) (a+b)
#define square(x) ((x)*(x))

main()

{
int a=3,b=4;
printf("Sum is %d",sum(a,b));
printf("Square is %d",square(sum(a,b)));

}
```

Function Basics : *Function header, function prototype specification, function definition and function call. Actual, formal parameters and return statement. Examples – standard library functions and simple user defined functions. Scope and extent of variables.*

Storage classes – *automatic, register, static and global.*

Function Basics

A program consists one or more functions, one of them being main() function. main() is a user defined function. Functions are self contained program structure designed with a single purpose.

A bigger C program is divided into small building blocks called C function. Collection of these functions creates a C program.

To solve Complex problems we divide it into smaller sub problems, these sub problems are implemented in c with functions.

There are 2 types of functions in C. They are, 1. Library functions 2. User defined functions

Function definition

A function is a set of statements that performs a specific and well defined task.

Function definition has the following general form

```
<return type> function-name (<parameter list>) //function header
{
  declaration statements;
  Executable statements ; //function body
}
```

Function header and function body are the basic elements of a function. The header provides return type, function name and number of input parameters with their data type(the order of parameter is important). The body contains local declarations and executable statements. The return statement is evaluated and sent back to calling function. The expression in return is optional. If return expression is omitted the control goes back to the calling function without returned value.

Only one value is returned to the calling function.

Function should be declared before they are used. Otherwise a prototype must be written.

Function Prototype

A function prototype tells the compiler the number and type of arguments that are to passed to function and the type of value that is to be returned by the function. The void is used if no value is returned by the function.

Example : int sum(int,int);

Uses of functions

- C functions are used to avoid rewriting same code again and again in a program.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be traced when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

Actual Parameter :

Parameter Written In Function Call is Called “Actual Parameter”.

```
void main()
{
int num1;
display(num1); // num1 is Actual parameter

}
void display(int para1)
{
-----
-----
}
```

Formal Parameter :

Parameter Written In Function Definition is Called “Formal Parameter”.

```
void main()
{
int num1;
display(num1);
}

void display(int para1) // formal parameter
{
-----
-----
}
```

Formal parameters are always variables, actual parameters do not have to be variables may be numbers, expressions, or even function calls as actual parameters. Here are some examples of valid actual parameters in the function call to calculate_bill:

```
bill = calculate_bill (25, 32, 27);
```

```
bill = calculate_bill (50+60, 25*2, 100-75);
```

```
bill = calculate_bill (fred, franny, (int) sqrt(25));
```

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.

Each formal argument is initialized with its corresponding actual argument.

Calling functions

A function can be called using the function name followed by parentheses.

Type of Functions

May be classified based on arguments and return value into following categories

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

Following program demonstrate the above four types of functions

```
#include <stdio.h>
#define p 3.14
void area1(void) // first function type;
float area2(void) // second type;
void area3(int) // third function type;
float area4(int) // fourth function type;

void area1(void) //input in function, processing in function, output in function
{
int r;
float area;
printf("\n Enter the radius value : ");
scanf("%d",&r);
area=2*p*r*r;
printf("\n The area is : %f",area);
}

float area2(void) //input in function, processing in function, output in main
{
int r;
float area;
printf("\n Enter the radius value : ");
scanf("%d",&r);
area=2*p*r*r;
return area;
}

void area3(int r) //input in main, processing in function, output in function
{
float area;
area=2*p*r*r;
printf("\n The area is : %f",area);
```

```

}

float area4(int r) //input in main, processing in function, output in main

{
float area;
area=2*p*r*r;
return area;
}

void main()
{
int r;
float area;
area1(); // first type function call

area=area2(); //second type function call;

printf("\n The area is : %f",area);
printf("\n Enter the radius value : ");
scanf("%d",&r);
area2(r); //Third type function call
//Third type function call;
printf("\n Enter the radius value : ");
scanf("%d",&r);
area=area2(r); //Third type function call
printf("\n The area is : %f",area);
}
    
```

The C contains the some following library functions:

<code>printf</code>	Formatted Write
<code>scanf</code>	Formatted Read
<code><u>getchar</u></code>	Read Character
<code><u>gets</u></code>	Read String
<code><u>putchar</u></code>	Write Character
<code><u>puts</u></code>	Write String
<code><u>strcmp</u></code>	String Compare
<code><u>strcat</u></code>	String Concatenation
<code><u>strcpy</u></code>	String Copy
<code><u>strchr</u></code>	Search String for Character
<code><u>strstr</u></code>	Search String for Substring

<u>strlen</u>	String Length
<u>isalnum</u>	Test for Alphanumeric
<u>isalpha</u>	Test for Alphabetic
<u>isctrl</u>	Test for Control Character
<u>isdigit</u>	Test for Digit
<u>islower</u>	Test for Lowercase Letter
<u>ispunct</u>	Test for Punctuation Character
<u>isupper</u>	Test for Uppercase Letter
<u>isxdigit</u>	Test for Hexadecimal Digit
<u>tolower</u>	Convert to Lowercase
<u>toupper</u>	Convert to Uppercase
<u>fabs</u>	Absolute Value of Floating-Point Number
<u>ceil</u>	Ceiling
<u>floor</u>	Floor
<u>fmod</u>	Floating Modulus
<u>log</u>	Natural Logarithm
<u>log10</u>	Common Logarithm
<u>modf</u>	Split into Integer and Fractional Parts
<u>pow</u>	Power
<u>sqrt</u>	Square Root

Storage class

Storage class defines

- 1) the life span of the variable and
- 2) Scope (availability) of the variable.

The life span of variable specifies the duration for which the variable will retain a value.

Scope of a variable indicates the part of the program where the variable is usable. A variable could be available to a block of statements, a function, a program.

auto

Syntax : [storage class] <data type> variable name;

Variables declared within a function are automatic by default. auto variables defined in different functions will be independent of one another, even though they may have the same name.

Scope and life span is with in the function

Example : auto int a,b;

extern

One method of transmitting data across blocks and functions is to use external variables. Their scope is extended from two or more functions, and often an entire program. They are often referred to as global variables and are initialized to zero by default. They can be accessed from any function that falls within their scope. The following programs illustrate storage class extern, generates same output.

<pre>#include <stdio.h> int y; main() { void func(void); printf("in main %d",y); func(); printf("in main %d",y); } void func() { y++; printf("in func %d",y); }</pre>	<pre>#include <stdio.h> main() { extern int y; void func(void); printf("in main %d",y); func(); printf("in main %d",y); } Int y; void func() { y++; printf("in func %d",y); }</pre>
---	---

static

Static variables retain their values throughout the life of the program. They are initialized to 0 for integers and space for character variables.

<pre>#include <stdio.h> main() {int i; void func(void); i= func(); printf(" %d",i); i= func(); printf(" %d",i); i= func(); printf(" %d",i); } void func() { static int k=0; k++;return k; }</pre> <p>Output : 1 2 3</p>	<pre>#include <stdio.h> main() {int i; void func(void); i= func(); printf(" %d",i); i= func(); printf(" %d",i); i= func(); printf(" %d",i); } void func() { auto int k=0; k++;return k; }</pre> <p>Output : 1 1 1</p>
---	---

register

The register storage class tells the compiler that the variables are stored in high speed registers.If the compiler cannot allocate registers then the storage class is auto. The execution time can be reduced if values are stored in registers.

Comparison of storage classes

Features	Automatic Storage Class	Register Storage Class	Static Storage Class	External Storage Class
Keyword	auto	register	static	extern
Initial Value	Garbage	Garbage	Zero	Zero
Storage	Memory	CPU register	Memory	Memory
Scope	scope limited,local to block	scope limited,local to block	scope limited,local to block	Global
Life	Till the control remains within the block in which the variable is defined	Till the control remains within the block in which the variable is defined	value of variable persist between different function calls	As long as the program's execution doesn't come to an end
Memory location	Stack	Register memory	Segment	Segment
Use	General purpose use. Most widely used compared to other storage classes	Used extensively for loop counters	Used extensively for recursive functions	Used in case of variables which are being used by almost all the functions in a program

C program to Check Prime and Armstrong Number

```

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

int checkPrimeNumber(int n);
int checkArmstrongNumber(int n);

int main()
{
    int n, flag;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    // Check prime number
    flag = checkPrimeNumber(n);
    if (flag == 1)
        printf("%d is a prime number.\n", n);
    else
        printf("%d is not a prime number.\n", n);

    // Check Armstrong number
    flag = checkArmstrongNumber(n);
    
```

```

    if (flag == 1)
        printf("%d is an Armstrong number.", n);
    else
        printf("%d is not an Armstrong number.",n);
    return 0;
}

```

```

int checkPrimeNumber(int n)
{
    int I;
    for(i=2; i<=n/2; ++i)
    {
        // condition for non-prime number
        if(n%i == 0)
        {
            return 0;
        }
    }
    return 1;
}

```

```

int checkArmstrongNumber(int number)
{
    int originalNumber, remainder, result = 0;

    originalNumber = number;

    while (originalNumber != 0)
    {
        remainder = originalNumber%10;
        result += pow(remainder, 3);
        originalNumber /= 10;
    }

    // condition for Armstrong number
    if(result == number)
        flag = 1;
    else
        flag = 0;

    return flag;
}

```

Find reverse of a number using function

```

/*c program for reverse number using user define function- rev */
#include<stdio.h>
int rev(int );
int main()

```

```

{
int num,res;
printf("Enter any number : ");
scanf("%d", &num);
res = rev(num);
printf("Reversed number = %d",res);
return 0;
}

```

```

int rev(int n)
{
int r=0;
for(; n>=1; n=n/10)
    r = r*10 + n%10;
return r;
}

```

What is the output of the following following code

```

void main()

{
int a=5;
    {
        int a=6,b=7;
        printf(“%d %d”,a,b);
    }
printf(“%d”,a);
}

```

Program to find GCD and LCM using functions

```

#include<stdio.h>
void gcd(int,int);
void lcm(int,int);

int main()
{
    int a,b;

    printf("Enter two numbers: \t");
    scanf("%d %d",&a,&b);

    lcm(a,b);
    gcd(a,b);
    return 0;
}

//function to calculate l.c.m
void lcm(int a,int b)

```

```

{
    int m,n;

    m=a;
    n=b;

    while(m!=n)
    {
        if(m<n)
            m=m+a;
        else
            n=n+b;
    }

    printf("\nL.C.M of %d & %d is %d",a,b,m);
}

//function to calculate g.c.d
void gcd(int a,int b)
{
    int m,n;

    m=a;
    n=b;

    while(m!=n)
    {
        if(m>n)
            m=m-n;
        else
            n=n-m;
    }

    printf("\nG.C.D of %d & %d is %d",a,b,m);
}
    
```

Program to find gcd of three numbers

```

#include<stdio.h>
int gcd(int m,int n)
{
    int rem;
    while(n!=0)
    {
        rem=m%n;
        m=n;
        n=rem;
    }
    return(m);
}
    
```

```

}
main()
{
    int num1,num2,num3,gcd1,gcd2;
    printf("Enter three positive integers");
    scanf("%d%d%d",&num1,&num2,&num3);
    gcd1=gcd(num1,num2);
    gcd2=gcd(num3,gcd1);
    printf("\n GCD of %d %d %d is : %d\n",num1,num2,num3,gcd2);
}

```

Write a c program to Find sum of 1-2+3-4+....n

```

#include <stdio.h>
void main()
{
    int n,sum=0,s=1,t;
    printf("\nEnter a number : ");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        t=i*s;
        sum=sum+t;
        s=s*(-1);
    }
    printf("\n sum of series is %d",sum);
}

```

Program to Evaluate the series $1+x^3/3!-x^5/5!+x^7/7!+...$

```

#include <stdio.h>
#include<math.h>
long fact(int n)
{
    long f=1;
    int i;
    for(i=1;i<=n;i++)
    f=f*i;
    return f;
}

void main()
{
    int i,n,x,s=1;
    double term,res=1.0;

    printf("\nEnter x value number : ");
    scanf("%d",&x);
    printf("\nEnter no of terms : ");

```

```
scanf("%d",&n);

for(i=3;i<=n;i=i+2)
{
    term=s*pow(x,i)/(float)fact(i);
    res=res+term;
    s=s*(-1);
}
printf("\n sum of series is %lf",res);
}
```

Find The value of j

```
int incr (int i)
{
    static int count = 0;
    count = count + i;
    return (count);
}
main ()
{
    int i,j;
    for (i = 0; i <=4; i++)
        j = incr(i);
}
```

(a) 10 (b) 4 (c) 6 (d) 7 Answer (a)

Consider the following C function definition:

```
int Trial (int a, int b, int c)
{
    if ((a >= b) && (c < b)) return b;
    else if (a >= b) return Trial (a,c,b);
    else return Trial (b,a,c);
}
```

The function Trial:

- (a) Finds the maximum of a, b, and c
- (b) Finds the minimum of a, b and c
- (c) Finds the middle number of a, b, c
- (d) None of the above

Ans: (c)

Consider the following C program

```
int a, b, c = 0;
void prtFun (void);
int main ()
{
    static int a = 1; /* line 1 */
```



```

prtFun();
a += 1;
prtFun();
printf ( "\n %d %d " , a, b );
}

```

```

void prtFun (void)
{
    static int a = 2; /* line 2 */
    int b = 1;
    a += ++b;
    printf ( " \n %d %d " , a, b);
}

```

What output will be generated by the given code segment?

(A) 3 1

4 1

4 2

(B) 4 2

6 1

6 1

(C) 4 2

6 2

2 0

(D) 3 1

5 2

5 2

Answer (C)

What output will be generated by the given code segment if:

Line 1 is replaced by “auto int a = 1;”

Line 2 is replaced by “register int a = 2;”

(A) 3 1

4 1

4 2

(B) 4 2

6 1

6 1

(C) 4 2

6 2

2 0

(D) 4 2

4 2

2 0

Answer (D)

// C program to convert decimal numbers to binary numbers

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

long decimalToBinary(long n);
int main() {
    long decimal;
    printf("Enter a decimal number\n");
    scanf("%ld", &decimal);
    printf("Binary number of %ld is %ld", decimal, decimalToBinary(decimal));

    return 0;
}

/* Function to convert a decimal number to binary number */
long decimalToBinary(long n) {
    int remainder;
    long binary = 0, i = 1;

    while(n != 0) {
        remainder = n%2;
        n = n/2;
        binary= binary + (remainder*i);
        i = i*10;
    }
    return binary;
}
```

Unit – IV

Arrays: Concepts, arrays representation in c, passing array as argument to functions. Two – dimensional array – 2D array organization in memory- row major and column major, Basics of multidimensional arrays. Examples programs on Matrix operations.

Pointers : Introduction (Basic concepts), Pointers as arguments to functions, pointer to arrays, pointer-to-pointer, array-of-pointers, Dynamic memory allocation functions, programming applications.

Arrays

Ordinary variable are used to store one value. If there were a large amount of similar data to be handled, then using a different variable for each data would become difficult. For example to store and calculate average marks of ten subjects marks for one student we need ten variables to store ten subject marks. If data is more still more variables required. c provides a simple solution is to define a single variable to store all ten subjects marks.

c permits us to represent a list of values with one variable called array. Arrays are defined as much same manner as ordinary variables, except that each array name must be accompanied by the size of the array.

Syntax to declare a one dimensional array : <data type > <array name>[size];

The size indicates maximum number of elements that can be stored in the array.

Examples :

```
int a[10]; // a is an array of type integer with the size 10, we can store 10 integer
           // values.
float b[20]; // b is an array of type float with the size 20, we can store 20 float
            //values.
char c[15]; // c is an array of type character with the size 15, we can store 15
           //characters.
```

We can use the array element by writing array name along with the subscript in square brackets. For example a=arr[2] expression assigns third element of array into variable a. The lower limit of an array is 0 and upper limit is size-1.

Following is the program to accept 3 subjects marks of a student and calculates the average:

```
#include<stdio.h>
void main()
{
int a[3]; // a is an array to store 3 subjects marks.
float avg;
printf("\n Enter three subjects marks : ");
scanf("%d%d%d",&a[0],&a[1],&a[2]);
```

```
avg=(a[0]+a[1]+a[2])/3.0;
printf("\n The average marks : %f",avg);
}
```

in the above program 'a' is a one dimensional array and 'a' is the base address or first byte address of the array. a[0] represents first value, a[1] represents second value and a[2] represents third value. Suppose the starting address of a[0] is 2120. Then, the next address of a[1] will be 2124, address of a[2] will be 2128 and so on. It's because the size of a float is 4 bytes.

Initialization of one dimensional array

Syntax :

```
<data type > <array name>[size]={val1,valw,val3...};
```

The array can be initialized at the time of declaration. If the number of values with in the braces are less than the array size, then that many array elements will be initialized, the remaining elements will be set to zero.

The array size need not be specified explicitly when initial values are mentioned.

```
int a[5]={0}; // all values set to zero
float b[5] = {1.2,2.5,3.0};
int c[ ]={1,2,4,8}; // array size is 4
```

Passing array as argument to functions

Like constants and variables it is also possible to pass the value of an array element or even entire array as an argument to a function. To pass array element to a function it is similar to pass a value or a variable.

The following program demonstrate the passing of an array element as argument to a function

```
#include<stdio.h>
#define n 10
int arr_sum(int x );
void main()
{
int i;
int a[n]={1,2,3,4,5,6,7,8,9,10},i;
i=arr_even(a[3]);
if(i==0)
printf("Number is Even);
else
printf("Number is odd);

}

int arr_even(int x)
{
```

```

if(x%2==0)
return 0;
return 1;
}
    
```

The following program demonstrate the passing of an array as argument to a function

```

#include<stdio.h>
#define n 10
int arr_sum(int x[],int );
void main()
{
int a[n]={1,2,3,4,5,6,7,8,9,10},i;
printf("Sum is %d",arr_sum(a,n);
}

int arr_sum(int x[],int size)
{
int i,sum=0;
for(i=0;i<n;i++) sum=sum+x[i];
return(sum);
}
    
```

When an array is passed as an argument to a function, the base address or starting byte address of the array is passed. The array name is written with a pair of empty square brackets. The size of the array is not mentioned in square brackets. The size of the array is specified as second argument to the function.

Write a program to find number of even values in the given list using function.

```

int eve(int a[ ],int s)
{
int tot=0,i
for(i=0;i<s;i++)
{
if(a[i]%2==0) tot++;
}
return tot;
}

main()
{
int x[10]={2,3,4,5,6,7,8,9,2,1};
printf("%d",eve(x,10);
}
    
```

Write a program to accept 15 characters and copy digits into first array, characters into second array and other symbols into third array.

Write a program to find the largest element in the array

```
#include<stdio.h>
main()
{
int a[10],i,max;
//accpt 10 elements
for(i=0;i<10;i++)
{
printf("\n Enter the value for position %d in the array ",i+1);
scanf("%d",&a[i]);
}
max=a[0];
for(i=1;i<10;i++)
{
if(a[i]>max) max=a[i];
}
printf("\n The largest element is %d",max);
}
```

Write a program to find the largest and smallest element in the array using functions.

Write a program to find mean, variance and standard deviation of 10 numbers.

```
main()
{
int a[10],i;
float mean,variance,sd,sum=0,sumsq=0;
//accpt 10 elements
for(i=0;i<10;i++)
{
printf("\n Enter the value for position %d in the array ",i+1);
scanf("%d",&a[i]);
}
for(i=1;i<10;i++) sum=sum+a[i];
mean=sum/10.0;
for(i=1;i<10;i++) sumsq=sumsq+pow((mean-a[i]),2);
variance=sumsq/10.0;
sd=sqrt(variance);
printf("Mean : %f, Variance : %f, Std.Deviation :%f",mean,variance,sd);
}
```

Two Dimensional array

In one dimensional array the array elements are stored contiguously one after the other which is called linear way. When it is required to store a list of values we use one dimensional array, when it is required to store values of similar type in the form of table which contains rows and columns we use two dimensional array.

The elements of two dimensional array also are stored in the memory in a continuous manner – first row elements, second row elements and so on. The elements have contiguous memory storage locations. The elements are arrange in row – major, some systems uses column-major manner also.

Two dimensional array are declared the same way as one dimensional array like

```
int a[4][5]; // a is a two dimensional array of 4 rows and 5 columns of type int
float b[4][5]; // b is a two dimensional array of 4 rows and 5 columns of type float
char c[4][4]; // c is a two dimensional array of 4 rows and 5 columns of type char
```

The following program prints array values and memory locations

```
main()
{
int a[2][3]={1,2,3,4,5,6};
//printing values
for (i=0;i<2;i++) // outer loop for rows
{
for (j=0;j<3;j++) // inner loop for columns
printf(“%d “,a[i][j]);
printf(“\n”); // each row in one line
}
//printing address of each location
for (i=0;i<2;i++) // outer loop for rows
{
for (j=0;j<3;j++) // inner loop for columns
printf(“%u “,&a[i][j]);
printf(“\n”); // each row in one line
}
}
```

Initialization of two dimensional array

The array elements initially contain garbage values with auto storage class and with static storage class array elements contain zeros.

```
static int a[2][3];

int a[3][4] = {0} // all values with zero
int a[3][4] = {1,2,3,4,5,6,7,8} //first row with 1,2,3,4 second row with 5,6,7,8
//remaining with zeros
int a[3][4] = {{1,2},{4,5,6,8},{7}} //first row with 1,2 and 0,0 second row with 4,5,6,8
//and third row with 7,0,0,0.
int a[][4]= {1,2,3,4,5,6,7,8} //first row with 1,2,3,4 second row with 5,6,7,8 and
//here row size is 2.
```

2D array organization in memory

Consider a 3×3 matrix like this:

A11 A12 A13
 A21 A22 A23
 A31 A32 A33

The elements have to be stored linearly in the memory space, we have many possible ways to store them. Here are some of the possibilities

Row Major Ordering , Column Major Ordering

Row Major method

- In this method, the elements of an array are filled up row-by-row such that the first row elements are stored first, then the second row and so on.
- Most of the high level programming languages like C/C++, Java, Pascal, etc use this method for storing multidimensional arrays.
 1. A11 A12 A13 A21 A22 A23 A31 A32 A33

Column Major method

- In Column Major ordering, all the elements of the first column are stored first, then the next column elements and so on till we are left with no columns in our 2-D array.
 2. A11 A21 A31 A12 A22 A32 A13 A23 A33

Passing two dimensional array to a function

Following is the program to demonstrate how to pass a two dimensional array to a function

```
#include<stdio.h>
int largest(int a[ ][5],int m, int n); // Function prototype
mai()
{
int a[3][5]={{45,60,30,75,55},{20,30,80,95,65},{25,70,10,75,90}};
printf("\n The largest element is %d",largest(a,3,5); // Function call
}
int largest(int a[ ][5],int ,int n) // Function header
{
int i,j,max=0;
for(i=0;i<m;i++)
for(j=0;j<n;j++)
if(a[i][j]>max) max=a[i][j]);
return max;
}
```

The matrix name a is followed by two pairs of brackets. The number of rows need not be given in the first pair of brackets, the second pair contains 5 that provides the explicit number of columns.

consider the following program fragment. What value is assigned to sum.

```
int x[ ][3]={1,2,3,4,5,6,7,8,9,10,11,12};
sum=x[1][1]+x[2][2];
```


The annual examination is conducted for 50 students for three subjects. Write a program to read the data and determine for following

- a) Total marks obtained by each student
- b) The highest marks in each subject and the roll number of the student who secured it
- c) The student who obtained the highest total marks

We can represent the student examination data as a two dimensional array of 50 rows and 5 columns. First column denoted the students roll number., next three columns stores three subject marks and fifth column fills the total marks of three subjects. The declaration and schematic representation (with sample data) of the array as follows –

```
int exam[50][5];
```

Roll No	Marks in subject-1	Marks in subject-2	Marks in subject-3	Total Marks
101	75	67	78	
102	80	76	82	
103	88	78	76	
...	
...	
...	
148	67	78	66	
149	78	98	76	
150	87	89	67	

Note : Total marks will be computed by program and filled

For example exam[2][0] gives the roll no 103. The array element exam[2][1], exam[2][2],exam[2][3] specify the marks in subject-1,subject-2 and subject-3 obtained by the student with roll number 103.

```
#include<stdio.h>
#define N 50
main()
{
static int exam [N][5];
int i,j;
int max1,max2,max3; // Highest marks in three subjects
int m1,m2,m3 // Roll No of students with highest marks in sub 1, sub 2 , sub3
int max_m; //Roll no of student who got highest total marks
int max; // highest total marks
max=max1=max2=max3=0;

for(i=0;i<N;i++) // Read roll no and marks in three subjects
{
for(j=0;j<4;j++)
scanf("%d",&exam[i][j]);
```

```

exam[i][4]=exam[i][1]+ exam[i][2]+ exam[i][3];// Total marks filling
}

for(i=0;i<N;i++)
{
    if(exam[i][1]>max1){ max1=exam[i][1]; m1=exam[i][0];}
    if(exam[i][2]>max2){ max2=exam[i][2]; m2=exam[i][0];}
    if(exam[i][3]>max3){ max3=exam[i][3]; m3=exam[i][0];}

    if(exam[i][4]>max){ max=exam[i][4]; max_m=exam[i][0];}
}
printf("\nHighest marks ins sub-1 =%d, Roll No=%d",max1,m1);
printf("\nHighest marks ins sub-2 =%d, Roll No=%d",max2,m2);
printf("\nHighest marks ins sub-3 =%d, Roll No=%d",max3,m3);
printf("\nHighest Total marks =%d, Roll No=%d",max,max_m);
}

```

Multi dimensional array

Multi dimensional array can have three, four or more dimensions. The first dimension is called plane which consists of rows and columns. The three dimensional array to be an array of two dimensional arrays. It considers the two dimensional array to be an array of one dimensional arrays.

We can initialize a three dimensional array in a similar way like a two dimensional array. Here's an example,

```

int test[2][3][4] = {
    { //plane 0
        {3, 4, 2, 3}, //row 0
        {0, -3, 9, 11}, //row 1
        {23, 12, 23, 2} //row 2
    },
    { //plane 1
        {13, 4, 56, 3}, //row 0
        {5, 9, 3, 5}, // row1
        {3, 1, 4, 9} //row 2
    }
};

```

Program to perform matrix multiplication using function

```

#include<stdio.h>
#define M 10
#define N 10
void mul_mat(int A[M][N],int B[M][N],int C[M][N],int s,int t,int u,int v);
void main()
{
    int a[M][N],b[M][N],c[M][N];

```

```

int i,j;//for indexing
int m,n,p,q;
printf("\nEnter the rows and columns of first matrix:");
scanf("%d%d",&m,&n);
printf("\nEnter the rows and columns of second matrix");
scanf("%d%d",&p,&q);
printf("\nEnter the first matrix:");
for(i=0;i<m;i++)
    for(j=0;j<n;j++)
    {
        printf("\na[%d][%d]=",i+1,j+1);
        scanf("%d",&a[i][j]);
    }
printf("\nEnter the second matrix:\n");
for(i=0;i<p;i++)
for(j=0;j<q;j++)
{
printf("\nb[%d][%d]=",i+1,j+1);
scanf("%d",&b[i][j]);
}

mul_mat(a,b,c,m,n,p,q);//function calling

printf("\nThe resulting matrix:\n");
for(i=0;i<m;i++)
{
for(j=0;j<q;j++)
{
printf("%d\t",C[i][j]);
}
printf("\n");
}
}

void mul_mat(int A[M][N],int B[M][N],int C[M][N],int s,int t,int u,int v)
{
int i,j,k;
if(t!=u)
{
printf("\nNot possible");
exit(0);
}
else
{
for(i=0;i<s;i++)
{
for(j=0;j<v;j++)
{
C[i][j]=0;
for(k=0;k<v;k++)

```

```

        {
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
    }
}
}
}
}

```

Write a program to display transpose of a matrix

```

#include <stdio.h>

int main()
{
    int a[10][10], transpose[10][10], r, c, i, j;
    printf("Enter rows and columns of matrix: ");
    scanf("%d %d", &r, &c);

    // Storing elements of the matrix
    printf("\nEnter elements of matrix:\n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("Enter element a%d%d: ", i+1, j+1);
            scanf("%d", &a[i][j]);
        }

    // Displaying the matrix a[][] */
    printf("\nEnter Matrix: \n");
    for(i=0; i<r; ++i)
    {
        for(j=0; j<c; ++j)
        {
            printf("%d ", a[i][j]);
        }
        printf("\n\n");
    }

    // Finding the transpose of matrix a
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            transpose[j][i] = a[i][j];
        }

    // Displaying the transpose of matrix a
    printf("\nTranspose of Matrix:\n");
    for(i=0; i<c; ++i)
    {

```

```

for(j=0; j<r; ++j)
{
    printf("%d ",transpose[i][j]);
}
    printf("\n\n");
} return 0;}
    
```

Pointers

Pointers : Introduction (Basic concepts), Pointers as arguments to functions, pointer to arrays, pointer-to-pointer, array-of-pointers, Dynamic memory allocation functions, programming applications.

A pointer is a variable which is used to store the address of another variable. Like any variable or constant, you must declare a pointer before using it, to store any variable address. The general form of a pointer variable declaration is –

```

int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
    
```

The above are the few examples of pointer declarations. **If you need a pointer to store the address of integer variable then the data type of the pointer should be int.** Same case is with the other data types.

& operator is also known as “Address of” Operator.

```
printf("Address of var is: %u", &num);
```

Point to note: %u or %p is used for printing variable’s address. Above printf function display the address of a variable num but **how can you store that address in some other variable?** That’s where pointer is used.

Pointer is just like another variable, the main difference is that it stores address of another variable rather than a value.

- * Operator – “Value at Address” Operator
- * Operator is also known as indirection operator.

Lets take the same example with a pointer variable –

```

int main()
{
    /*Pointer of integer type*/
    int *p
    int var = 10;
    
```

```

/*Assigning the variable address to pointer*/
p= &var;

printf("Value of var is: %d", var);
printf("Address of var is: %u", p);
return 0;
}

#include <stdio.h>

int main () {

int var = 20; /* actual variable declaration */
int *ip; /* pointer variable declaration */

ip = &var; /* store address of var in pointer variable*/

printf("Address of var variable: %p\n", &var );

/* address stored in pointer variable */
printf("Address stored in ip variable: %p\n", ip );

/* access the value using the pointer */
printf("Value of var variable: %d\n", *ip ); // display var value i.e. is 20

return 0;
}

```

Let's assume that we need to add 1 to the variable a. We can do this with any of the following statements, assuming that the pointer p is initialized p=&a,

```
a++; a=a+1; *p=*p+1; (*p)++;
```

Accessing one variable with multiple pointers :

```

#include <stdio.h>

int main () {

int var = 20; /* actual variable declaration */
int *p,*q,*r; /* pointer variable declaration */

p=&a;
q=&a;
r=&a;

printf("a values with p %d\n", *p );
printf("a values with q %d\n", *q );
printf("a values with r %d\n", *r );
}

```

```

return 0;
}

```

Accessing multiple variable values with one pointer:

```

#include <stdio.h>

int main () {

    int a = 5,b=10; /* actual variable declaration */
    int *p;        /* pointer variable declaration */

    p=&a;
    printf("a values with p %d\n", *p );
    p=&b;
    printf("b values with p %d\n", *p );
    return 0;
}

```

Pointers as arguments to functions

We can pass data to the called function and we also can pass address(pointers). To pass addresses the formal parameters in the called function are defined as a pointer to variables.

Following is the program which changes value in the function using pointer passing through the function

```

void chng(int *a)
{
return(*a+1);
}
void main()
{
int b=5;
printf("\n b value before function call is %d",b); // b value 5
chng(&b);
printf("\n b value after function call is %d",b); // ba value is 6
}

```

call by value and call by reference

Call by value essentially means that a copy of the variable is passed into the function. The function does not modify the original. Pass by reference means that essentially the variable address is passed (though the name may change).

Example using Call by Value

```

#include <stdio.h>

void swapByValue(int, int); /* Prototype */

```

```
int main() /* Main function */
{
    int n1 = 10, n2 = 20;

    /* actual arguments will be as it is */
    swapByValue(n1, n2);
    printf("n1: %d, n2: %d\n", n1, n2);
}
```

```
void swapByValue(int a, int b)
{
    int t;
    t = a; a = b; b = t;
}
```

OUTPUT

```
=====
n1: 10, n2: 20
```

Program 19:

Example using Call by Reference

#include <stdio.h>

void swapByReference(int*, int*); /* Prototype */

```
int main() /* Main function */
{
    int n1 = 10, n2 = 20;

    /* actual arguments will be altered */
    swapByReference(&n1, &n2);
    printf("n1: %d, n2: %d\n", n1, n2);
}
```

```
void swapByReference(int *a, int *b)
{
    int t;
    t = *a; *a = *b; *b = t;
}
```

OUTPUT

```
=====
n1: 20, n2: 10
```

Function returning a pointer

Program 20:

```
int * smaller(int *a, int *b)
{
    return (*a < *b?a:b);
}
```



```

void main()
{
int a,b,*ip;
scanf("%d%d",&a,&b);
ip=smaller(&a,&b);
printf("\n smaller value is %d",*ip);
}
    
```

Pointer to pointer

pointer points to an address of another pointer. Such pointers are known as **pointer to pointer**

How to declare a pointer to Pointer in C?

```
int **pr;
```

Here in this example pr1 is a pointer to pointer

Program 21:

```

#include <stdio.h>
int main()
{
    int num=123;

    /*Pointer for num*/
    int *pr2;

    /*Double pointer for pr2*/
    int **pr1;

    /* I'm reading the address of variable num and
     * storing it in pointer pr2*/
    pr2 = &num;

    /* storing the address of pointer pr2 into another pointer pr1*/
    pr1 = &pr2;

    /* Possible ways to find value of variable num*/
    printf("\n Value of num is: %d", num);
    printf("\n Value of num using pr2 is: %d", *pr2);
    printf("\n Value of num using pr1 is: %d", **pr1);

    /*Possible ways to find address of num*/
    printf("\n Address of num is: %u", &num);
    printf("\n Address of num using pr2 is: %u", pr2);
    printf("\n Address of num using pr1 is: %u", *pr1);

    /*Find value of pointer*/
    printf("\n Value of Pointer pr2 is: %u", pr2);
    printf("\n Value of Pointer pr2 using pr1 is: %u", *pr1);

    /*Ways to find address of pointer*/
    
```

```

printf("\n Address of Pointer pr2 is:%u",&pr2);
printf("\n Address of Pointer pr2 using pr1 is:%u",*pr1);

/*pointer to pointer value and address*/
printf("\n Value of Pointer pr1 is:%u",pr1);
printf("\n Address of Pointer pr1 is:%u",&pr1);

return 0;
}

```

Arrays and pointers

There is strong relation between pointers and arrays. Array name is the base address of the array or first element address, we can assign array address into a pointer. Any operation by arrays can also be done with pointers.

The following declaration defines an array of size 10, that is the block of 10 consecutive elements are a[0],a[1],a[2]...a[9]

```
int a[10];
```

Let us assume that an integer occupy 2 bytes of memory and base address of array is 9700, so first value a[0] is at address 9700, next value a[1] is at 9702, a[2] is at 9704 all elements of the array are located adjacent to each other, if data elements are 5,10,15,..50 is shown as

The notation a[i] refers to the ith element,
pa is a pointer to an integer declared as int *pa;

then the assignment
pa=&a[0];

sets the pa to first element of a, that is pa contains address of a[0] base address of a

The assignment x=*pa; will copy the content of a[0] into x;

Program to reads array elements and finds the sum using pointer

```

#include<stdio.h>
void main()
{
int a[10],*pa,i,n=10,sum=0;

pa=a;
for(i=0;i<n;i++)
{
scanf("%d",pa+i);
}
for(i=0;i<n;i++)
{

```

```
sum=sum+*(pa+i);
}
printf(“\n %d”,sum);
}
```

Operations on pointers

Following operations are possible on pointer variables

- a) Pointer arithmetic
- b) Pointer comparison

We can perform addition and subtraction on pointers.

consider `int s,*px; px=&x; x=25;`

Assume the memory address of x is 5003, if `px=px+3;` is executed then the px will contain 5009 is $5003+3*(\text{size of integer})$ i.e. $5003+3*2=5009$

i.e. base address + number * size of datatype

```
++*px; // increments the value of variable x by 1
--*px // decrement x by 1
*px +10 // add 10 to x
*px*10+x // is same as x*10+x
```

Two pointers can be compared with a relational operator.

```
int x,y,*px,*py;
px=&x;
py=&y;
x=25;
y=100;
if x is at address 8686, and y is at address 8688
px>py // is false
py>=px // true
px+1==py //true
px-py >0 //false , its value is -1
```

```
int array[5]; /* Declares 5 contiguous integers */
int *ptr = array; /* Arrays can be used as pointers */
ptr[0] = 1; /* Pointers can be indexed with array syntax */
*(array + 1) = 2; /* Arrays can be dereferenced with pointer syntax */
*(1 + array) = 2; /* Pointer addition is commutative */
2[array] = 4; /* Subscript operator is commutative */
```

Pointer Notation	Array Notation	Description
<code>**ptr</code>	<code>*array[]</code>	Declares an array of pointers.
<code>*ptr</code>	<code>array[0]</code>	The address of the first pointer in the array; for a

		String array, the first string.
*(ptr+0)	array[0]	The same as the preceding entry.
**ptr	array[0][0]	The first element of the first pointer in the array; The first character of the first string in the array.
***(ptr+1)	array[1][0]	The first element of the second pointer in the array; The first character of the second string.

DYNAMIC MEMORY ALLOCATION IN C:

The process of allocating memory during program execution is called dynamic memory allocation.

Dynamic Memory Allocation functions in c

C language offers 4 dynamic memory allocation functions. They are,

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

malloc function

- malloc () function is used to allocate space in memory during the execution of the program.
- malloc () does not initialize the memory allocated during execution. It carries garbage value.
- malloc () function returns null pointer if it couldn't able to allocate requested amount of memory.

Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

Here, *ptr* is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc function

- calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn't.

The name calloc stands for "contiguous allocation".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

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

This statement will allocate contiguous space in memory for an array of n elements. For example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

realloc function

realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.

If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

free function

free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

Example: Using C malloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

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

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

```
int main()
```

```
{
```

```
    int num, i, *ptr, sum = 0;
```

```
    printf("Enter number of elements: ");
```

```
    scanf("%d", &num);
```

```
    ptr = (int*) malloc(num * sizeof(int)); //memory allocated using malloc
```

```
    printf("Enter elements of array: ");
```

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

```
    {
```

```
        scanf("%d", ptr + i);
```

```
        sum += *(ptr + i);
```

```
    }
```

```
    printf("Sum = %d", sum);
```

```
    free(ptr);
```

```
    return 0;
```

```
}

```

C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

```
ptr = realloc(ptr, newsize);
```

Here, *ptr* is reallocated with size of newsize.

Example : Using realloc()

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

int main()
{
    int *ptr, i, n1, n2;
    printf("Enter size of array: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Address of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
        printf("%u\t", ptr + i);

    printf("\nEnter new size of array: ");
    scanf("%d", &n2);
    ptr = realloc(ptr, n2);
    for(i = 0; i < n2; ++i)
        printf("%u\t", ptr + i);
    return 0;
}
```

Array of pointers

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer –

```
int *ptr[5];
```

It declares ptr as an array of 5 integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows –

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

```
const int MAX = 3;
```

```

int main () {

    int var[] = { 10, 100, 200};
    int i, *ptr[MAX];

    for ( i = 0; i < MAX; i++) {
        ptr[i] = &var[i]; /* assign the address of integer. */
    }

    for ( i = 0; i < MAX; i++) {
        printf("Value of var[%d] = %d\n", i, *ptr[i] );
    }

    return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

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

```

Pointer to an array

Syntax:

```
data_type (*var_name)[size_of_array];
```

Example:

```
int (*ptr)[10];
```

Here *ptr* is pointer that can point to an array of 10 integers. Since subscript have higher precedence than indirection, it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of *ptr* is ‘pointer to an array of 10 integers’.

```

// C program to understand difference between
// pointer to an integer and pointer to an array of integers.
#include<stdio.h>

```

```

int main()
{
    // Pointer to an integer
    int *p;

    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int arr[5];

    // Points to 0th element of the arr.
    p = arr;

    // Points to the whole array arr.
    ptr = &arr;

    printf("p = %p, ptr = %p\n", p, ptr);
}

```

```

p++;
ptr++;

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

return 0;
}

```

Example questions

1) Consider the following declaration of a 'two-dimensional array in C:

```
char a[100][100];
```

Assuming that the main memory is byte-addressable and that the array is stored starting from memory address 0, the address of a[40][50] is

- (a) 4040 (b) 4050 (c) 5040 (d) 5050

Ans: option (b)

Explanation:

Address of a[40][50] = Starting Address + 40 x 100 x memorySize + 50 x memorySize
 $= 0 + 40 \times 100 \times 1 + 50 \times 1 = 4050$

Note: Since byte addressable, memorySize = 1

2) Assume the following C variable declaration

```
int *A [10], B[10][10];
```

Of the following expressions

- I. A[2] II. A[2][3] III. B[1] IV. B[2][3]

which will not give compile-time errors if used as left hand sides of assignment statements in a C program?

- (a) I, II, and IV only
 (b) II, III, and IV only
 (c) II and IV only
 (d) IV only

Ans: option (a)

Explanation:

A is an array of pointers, and A[2] can be used as left hand sides of assignment statements. Suppose we have another array of integers i.e. int marks[]={ 10,20,30,40}. Then we can assign A[2] = marks; Because marks represents the starting address of the array marks[], and on execution, the address is stored in the 3rd element of array A.

Considering the assignment A[2] = marks;, A[2][3] represents the element 40 (i.e. 4th element in the marks array). Therefore, A[2][3] also can be used as left hand sides of assignment statements. i.e. A[2][3] = 45; will make the contents of marks array as {10,20,30,45}.

B[2][3] =12; represents a simple assignment to an element of a two-dimensional array. So B[2][3] can also be used as left hand sides of assignment statements. B[1] cannot be used as left hand sides of assignment statements, because, since it is a two-dimensional array B[1] represents an address and we cannot write it.

3) What is the output of the following

```
#include<stdio.h>
void f(int *p, int *q)
{
    p = q;
    *p = 2;
}
int i = 0, j = 1;

int main()
{
    f(&i, &j);
    printf("%d %d \n", i, j);
    return 0;
}
(a) 2 2   (b) 2 1   (c) 0 1   (d) 0 2
```

Ans: option (d)

Explanation:

Initially i=0 and j=0

f(&i, &j); on execution p pointer points to i, and q pointer points to j.

p = q; means p now contains the address of j, therefore p also now points to j.

**p = 2;* 2 value is saved in the location where the p is pointing. Since p points to j, value of j is changed to 2 now.

Now returns to main function and prints i and j. Hence i=0 and j=2

4). What is printed by the following C program?

```
int f(int x, int *py, int **ppz)
{
    int y, z;
    **ppz += 1;
    z = **ppz;
    *py += 2;
    y = *py;
    x += 3;
    return x + y + z;
}

void main()
{
    int c, *b, **a;
```

```

c = 4;
b = &c;
a = &b;
printf( "%d", f(c,b,a));
}
    
```

(a) 18 (b) 19 (c) 21 (d) 22

Ans: option(b)

Hint: pointer b and py is pointing to c

Initially c = 4

ppz contains the address of b, therefore **ppz refers to the value to which b is pointing. Since b is pointing towards c, the value of c increments when **ppz += 1; statement executes.

Therefore c now becomes 5.

z=**ppz; means z = 5

py pointer is pointing towards c, therefore on execution of *py += 2;, value of c becomes 7.

y=*py; means y = 7

x contains value 4, therefore on execution of x+=3; x becomes 7.

5) The output of the following C program is

```

#include <stdio.h>
void f1(int a, int b)
{
    int c;
    c=a; a=b; b=c;
}

void f2(int *a, int *b)
{
    int c;
    c=*a; *a=*b;*b=c;
}

int main()
{
    int a=4, b=5, c=6;
    f1 (a, b);
    f2 (&b, &c);
    printf("%d", c-a-b);
}
    
```

Ans: -5

Explanation:

You can see that the f1 function is calls its parameters by value. Hence the modifications are made inside the scope of f1 functions does not effect any of the variables of main function.

But in f2, the parameters are called by reference.

In f2, we can see that pointer a points to variable b and pointer b points to variable c.

c = *a; // means that c = 5;

*a = *b; // means that *a = 6; that is b= 6;
 *b = c; // means *b = 5; that is c = 5;

6) What is the output of the following C code? Assume that the address of x is 2000 (in decimal) and an integer requires four bytes of memory.

```
int main()
{
    unsigned int x[4][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};
    printf("%u,%u, %u", x+3, *(x+3),*(x+2)+3);
}
```

- (a) 2036, 2036, 2036
- (b) 2012, 4, 2204
- (c) 2036, 10, 10
- (d) 2012, 4, 6

Ans: option (a)

Explanation:

Here x is a two-dimensional array. Also it is given that the base address of x = 2000. And the integer requires 4 bytes of memory. x[4][3] means that x has 4 rows and each row has 3 integers.

x + 3 means address of the 4th row (i.e. row with integers 10,11,12)
 = Base address of x + 3 * (size of memory required by a row)
 = 2000 + 3 * (3 * 4)
 = 2036

(x + 3) => Since x is a two-dimensional array one reference operator () will only return the starting address of the one-dimensional array. i.e. *(x + 3) will return the starting address of the 4th-row, which equals 2036.

*(x + 2) + 3 means the address of 4th-integer of 3rd-row [*(x + 2) means address of 3rd-row]
 => address of second row + 3 * (size of memory required by an integer)
 => address of second row + 3 * 4
 => 2024 + 12
 => 2036

Unit – V

Strings : String representation, String Input/output functions, Implementation of string manipulation functions.

Recursion: recursion- recursive functions and tracing recursive function calls
 Examples: Factorial, Fibonacci, Towers of Hanoi, Capability of main() to handle command line arguments.

A character is a single alphabet, number, punctuation mark or other symbol. A string is any sequence of characters. A string is an array of characters enclosed within a pair of double quotes. String can be of any length, the end of string will have null character denoted by '\0'. The ASCII value of NULL character is zero.

Strings

Syntax of declaring a character array

```
char <array name> [size];
```

Always size is one position more than number of characters in the string, because one position is for NULL character.

String can initialized in following ways

```
char str[12]="CVR COLLEGE";
char str[12]={ 'C','V','R',' ',' ','C','O','L','L','E','G','E','\0'};
char str[]="CVR COLLEGE";
char str[]={ 'C','V','R',' ',' ','C','O','L','L','E','G','E','\0'};
```

In the above last two declarations size is optional, the size assumed would be 12.

```
char str[]; // is invalid here size should be mentioned
char str[12]; // is vlid
str="CVR College"; // is invalid, str is base address and it is constant
```

Reading and Writing strings

scanf function to accept a string from keyboard using %s format specifier

```
main()
{
    char[25];
```

```
scanf("%s",str); // str is a base address so & is not required before str
printf("%s",str);
}
```

- 1) scanf accepts single word. scanf function is used to read string which contains sequence of characters without a white space.
- 2) gets function is used to input strings from the keyboard, gets can also read white spaces from the keyboard. To read multiple word string inputs we need to use multiple gets functions

```
syntax id gets(string);
```

The gets and scanf functions automatically add null character to the end of the string. puts and printf functions are used to display strings. syntax of puts is puts(string);

Operations on strings

The following examples demonstrate the common operations on strings , such as

- a) String length
- b) Concatenation of strings
- c) Copying one string to another
- d) Comparison of strings
- e) Extracting a substring from a string

Program to find length of string

The program reads a string using gets() function and using for loop the program scans character by character and counts the number of characters until a null character is encountered.

```
#include<stdio.h>
main()
{
char str[25],ch;int length,i=0;
puts("Enter a string");
gets(str);
length=0;
while( str[i]!='\0')
{i++;
length ++;
}
printf("\n string length is %d",length);
}
```

input/output is as follows :

Enter a String CVR College
string length is 11

Program to concatenate two strings

The program reads two strings and second string is joined to the first string. The program scans the first string character by character until a null character is encountered. Then the second string is scanned character by character until a null character is encountered, and each character of the second string is assigned to the first string starting from the end of the first string.

```
#include<stdio.h>
main()
{
char a[25],b[25];
int i,j;
i=0;
puts("Enter first string : ");gets(a);
puts("Enter second string : ");gets(b);
while( a[i]!='\0')
{i++;
}
j=0;
while( b[j]!='\0')
{
a[i]=b[j];
i++;j++;
}
a[i]=NULL;
printf("\n First string after concatenations is %s",a);
}
```

input/output is as follows :

Enter first String : star

Enter second string : wars

First string after concatenations is starwars

Program for copying one string to another string

```
#include<stdio.h>
main()
{
char str1[25],str2[25],ch; int i;
puts("Enter a string 1");gets(str1);
for(i=0;str[i]!='\0';i++)
str1[i]=str2[i];

str2[i]='\0';
printf("\n string 2 is %s",str2);
}
```

Comparison of strings

Programs scans the strings str1 and str2 character by character and compares a character of str1 with the character of str2, until a null character of either string is reached. If the characters are equal then the comparison continues otherwise the ASCII difference of two characters is computed . If the difference is zero then both strings are equal, if difference is negative str1 is smaller otherwise str2 is smaller

```
#include<stdio.h>
main()
{
char str1[25],str2[25],int i,diff=0;
puts("Enter first string : ");gets(str1);
puts("Enter second string : ");gets(str2);
for(i=0;str1[i]!='\0' || str2[i]!='\0';i++)
    if(str1[i]==str2[i]) continue;
else { diff=str1[i]-str2[i];
        break;
    }
if(diff>0) printf("str1 is greater");
    else if (diff<0) printf("str1 is smaller");
    else printf("both are same");
}
```

Extraction of a substring

A portion of a string is called substring

```
#include<stdio.h>
main()
{
char str[25]="Mahatma Gandhi");
char substr[24];
int i,j; // i indes for string, j index for substring
int start,length;
start=3,length=5;
for(i=start,j=0;j<length;i++,j++) substr[j]=str[i];
substr[j]='\0';
printf("\n substr : %s",substr);
}
```

String handling functions

c provides in-built functions to perform different operations on strings. To use string functions , the header file <string.h> is to be included.

- 1) strcat(string1, string2);


```
char string1[35]="Yahoo",string2[20]="Hotmail";
strcat(string1,string2);
strcat(string1,"Messanger");
```
- 2) strcpy(string1, string2);


```
char string1[35],string2[20]="Hotmail";
strcpy(string1,string2);
strcpy(string1,"Yahoo");
```
- 3) strlen(string1);


```
char string1[35]="CVR College of Engineering";int n;
n= strlen(string1);
```
- 4) strcmp(string1, string2);


```
char string1[35]="Yahoo",string2[20]="Hotmail";
```

the strcmp function returns zero if strin1=string2, returns negative value if strin1<string2 or positive value if strin1>string2

Pointers and strings

one way to allocate and initialize a string is to declare an array of characters as follow:

```
char str[]="Pointers and Strings";
```

We may declare same thing using pointer also

```
char *cptr="Pointers and Strings";
```

Both declarations re equivalent. In each case the compiler allocates enough space to hold the string along with its terminating null character.

A pointer pointing to a string contains the base address of the character array and array can be accessed using this pointer.

A program to demonstrate pointers with strings

```
main()
{
char str1[]="CVR College of Engineering";
char str2[]="Problem solving through c";
char *cptr;    //a pointer to type character
puts(str1);    // display CVR College of Engineering
puts(str2);    // display Problem solving through c

cptr=str1;    // base address of str1 is assigned to cptr
puts(cptr);    // display CVR College of Engineering
cptr=cptr+4;  // cptr advances and starts from str1[4]
puts(cptr);    // display College of Engineering

cptr=str2;    // base address of str2 is assigned to cptr
puts(cptr);    // display Problem solving through c
cptr=cptr+8;  // cptr advances and starts from str1[8]
puts(cptr);    // display solving through c
}
```

A program display the length of a string using pointer

```
#include<stdio.h>
main()
{
```

```

char str[25]="problem solving through c",*cptr;int length;
puts("Enter a string");
length=0;
cptr=str; // assigns base address to cptr
for(;*cptr!='\0';cptr++) length ++;
printf("\n string length is %d",length);
}
    
```

Command Line Arguments

main is function which is having parameters also, they are known as command line arguments. The parameters are passed to a function when a function call takes place, but how the parameters are passed to main as command line arguments is at the time of program is invocation.

main() function of a C program accepts arguments from command line they are

1. Number of arguments in the command line as integer
2. program name and user defined values as array of pointers to characters and number of array elements are equivalent to first argument number.

Program 10:

```

#include <stdio.h>
#include <stdlib.h>
int main(int ac, char *av[]) // command line arguments for five arguments
{
if(ac!=5)
{
printf("Arguments passed through command line not equal to 5");
return 1;
}

printf("\n Program name : %s \n", av[0]);
printf("1st arg : %s \n", av[1]);
printf("2nd arg : %s \n", av[2]);
printf("3rd arg : %s \n", av[3]);
printf("4th arg : %s \n", av[4]);
printf("5th arg : %s \n", av[5]);
return 0;
}
    
```

```

}
$./a.out maths english psc physics drawing
Program name : ./a.out
1st arg : maths
2nd arg : english
3rd arg : psc
4th arg : physics
5th arg : drawing
    
```

Recursion

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". For Example factorial of n is $n * \text{factorial of } n-1$ and so on.

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.

The following example calculates the factorial of a given number using a recursive function

```

#include <stdio.h>

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;
}
    
```

Program to generate Fibonacci series using recursion in c

```

#include<stdio.h>

void printFibonacci(int);

int main(){
    int k,n;
    
```

```

long int i=0,j=1,f;

printf("Enter the range of the Fibonacci series: ");
scanf("%d",&n);
printf("Fibonacci Series: ");
printf("%d %d ",0,1);
printFibonacci(n);
return 0;
}

```

```

void printFibonacci(int n){
    static long int first=0,second=1,sum;
    if(n>0){
        sum = first + second;
        first = second;
        second = sum;
        printf("%ld ",sum);
        printFibonacci(n-1);
    }
}

```

Sample output:

```

Enter the range of the Fibonacci series: 10
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34 55 89

```

Program 13: Sum of digits in c using recursion

```

#include<stdio.h>
int main(){
    int num,x;
    clrscr();
    printf("\nEnter a number: ");
    scanf("%d",&num);
    x=findsum(num);
    printf("Sum of the digits of %d is: %d",num,x);
    return 0;
}

```

```

int findsum(int n){
    if(n){
        r=n%10;
        s=s+r;
        findsum(n/10);
    }
    else
        return s;
}
    
```

GCD of Two Numbers using Recursion

```

#include <stdio.h>
int gcd(int n1, int n2);
int main()
{
    int n1, n2;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);

    printf("G.C.D of %d and %d is %d.", n1, n2, gcd(n1,n2));
    return 0;
}

int gcd(int n1, int n2)
{
    if (n2 != 0)
        return gcd(n2, n1%n2);
    else
        return n1;
}
    
```

Output

Enter two positive integers: 366 60
 G.C.D of 366 and 60 is 6.

Sum of Natural Numbers Using Recursion

```

#include <stdio.h>
int sum(int n);

int main()
{
    int num;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    printf("Sum = %d",sum(num));
    return 0;
}
    
```

```
int sum(int n)
{
    if(n != 0)
        return n + sum(n-1);
    else
        return n;
}
```

Output

Enter a positive integer: 3
Sum = 6

Initially, the sum() is called from the main() function with 3 passed as an argument.

The number 3 is added to the result of sum(2).

In the next function call from sum(2) , 2 is passed which is added to the result of sum(1). This process continues until n is equal to 0.

When n is equal to 0, there is no recursive call and this returns the sum of integers to the main()function.

When num is equal to 0, the if condition fails and the else part is executed returning the sum of integers to the main() function.

Advantages and Disadvantages of Recursion

Recursion makes program simple and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is important then, we should avoid using recursion. Recursions use more memory and are generally slow.

References

1. C Programming & Data Structures – Behrouz A.Forouzen, Richard F, Gilberg, CENGAGE Learning
2. C & Data Structures – V.V.Muniswamy, I.K.International
3. ANSI C Programming – Gary J.Bronson, CENGAGE Learning
4. [https://en.wikipedia.org/wiki/C_\(programming_language\)](https://en.wikipedia.org/wiki/C_(programming_language))
5. <https://www.tutorialspoint.com/cprogramming/>
6. <http://fresh2refresh.com/c-programming/>
7. www.indiabix.com/c-programming/questions-and-answers/
8. Course File – CVR Colege of Engineering