UNIVERSITY OF MADRAS

BCA I YEAR – 2014

C programming [let us make c easy]

Let us Make

c easy

Sanjeev Gautam

Welhams College

BCA I YEAR – 2014 BATCH

SYLLABUS [PROGRAMMING IN C]

MODULE - I

C fundamentals, Character Sets, Identifier and keywords, data types, constants, variables – Declarations, Expressions, Statements, Arithmetic, Unary, Relational and logical, Assignment and Conditional, Operators - Library functions.

MODULE – II

Data input output functions - Simple C programs – Flow of control if, if else, while, do- while, for loop, Nested control structures - Switch, break and continue, go to statements - Comma operator. Functions – Definition - proto-types - Passing arguments - Recursions. Storage Classes - Automatic, External, Static, Register Variables – Multi-file programs.

MODULE - III

Arrays - Defining and Processing - Passing arrays to functions – Multi-dimension arrays – Arrays and String. Structures - User defined data types - Passing structures to functions – Self referential structures – Unions - Bit wise operations. Pointers - Declarations - Passing pointers to Functions - Operation in Pointers - Pointer and Arrays - Arrays of Pointers - Structures and Pointers - Files: Creating, Processing, Opening and Closing a data file.

MODULE – IV [LAB]

Summation of Series: 1. Sin(x), 2. Cos(x), 3. Exp(x) (Comparison with built in functions)

String Manipulation: Counting the no. of vowels, consonants, words, white spaces in a line of text and array of lines, Reverse a string & check for palindrome, Substring detection, count and removal, Finding and replacing substrings

Recursion: nPr, nCr, GCD of two numbers, Fibonacci sequence, Maximum & Minimum, Towers of Hanoi.

Matrix Manipulation: Addition & Subtraction, Multiplication, Transpose, and trace of a matrix, Determinant of a Matrix

Sorting and Searching: Insertion Sort, Bubble Sort, Linear Search, Binary Search

BOOKS FOR REFERENCE

B.W. Kernighan and D.M.Ritchie, The C Programming Language, 2nd Edition, PHI, 1988.

H. Schildt, C: The Complete Reference, 4th Edition, TMH Edition, 2000

Gottfried,B.S, Programming with C, Second Edition, TMH Pub. Co. Ltd., New Delhi 1996.

Kanetkar Y., Let us C, BPB Pub., New Delhi, 1999.

WHAT ARE PROGRAMMING LANGUAGES?

Those languages which are used in the computer programming are called programming languages. Since the inception of computer, there are 3 types of computer programming languages which have been evolved:

1. Machine Language or Low Level Language
2. Assembly Level Language
3. High – Level Language

MACHINE LANGUAGE

- The computer language which is written in binary format (i.e. containing only 0's and 1's) as the set of instructions to the computer is called machine language.
- It is the only one language that is understood by the computer without any modification.

Disadvantages

It is seen that writing machine language programs is very bulky, time consuming process liable to errors and need to improve this system lad to the development of Assembly Language.

ASSEMBLY LANGUAGE

It combines the restricted use of machine language with mnemonics or human memory aids. These aids are abbreviated forms of certain functions like ADD for addition, SUB for subtraction and HLT for halt (exit) the program. This makes the programmer easy to remember the code of the functions. It was machining dependent language i.e. the code for one computer may not applicable for another computer system.

HIGH LEVEL LANGUAGE

The programming language which can be easily understood by human is called high level language. These languages transcribe the programs as statements, using a very limited vocabulary from English. Some of the popular and widely used high levels programming languages are:

FORTRAN, ALGOL (algorithm), COBOL, *BASIC*, *PASCAL*, C, C++, Java, Visual Basic etc.

Advantages over low level languages

- **User – friendly:** The use of 'English' with proper syntax made it easy for the user.
- Versatile and provide proper documentation.
- Programs written in high level language can be easily changed.

LANGUAGE TRANSLATOR

Computers understands only the language of 0's and 1's i.e. machine language. Therefore, other languages should be translated into machine language. Those computer software or programs which convert the program written in one language into another language are called language translators. There are 3 types of language translators which are as follows: Assembler, Compiler, Interpreter

ASSIGNMENT – I

¤ **What are the main differences between Assembler, Compiler and Interpreter?**
¤ **Define language translator.**
¤ **Computer understands the language of 0's and 1's. What does it means?**
¤ **Define :**
  - **Programs**
  - **Programming**
  - **Programmer**
  - **Programming languages.**

INTRODUCTION TO C-PROGRAMMING LANGUAGES

¬ C is a general-purpose, structured programming language that is powerful, efficient and compact.

¬ Its instructions consists of terms that resemble algebraic expressions, supplemented by certain English keywords such as if, else, for, and do.

¬ Used for writing system programs and application programs.

The root of all modern language is ALGOL, introduced in early 1960s. C was evolved from ALGOL, BCPL and B by Dennis Ritchie (father of the C language) at Bell Laborites in 1972. C uses many concept of this programming language and added the concept of data types and other powerful features. Since it was developed along the UNIX operating system, it is strongly associated with UNIX.

REASONS BEHIND THE POPULARITY OF THE C – PROGRAMMING [its features | characteristics]

Due to the following characteristics of C programming language, we need this programming language:

1) **Flexibility nature:**

   C program is a flexible language because it includes only the words of the English language which can be easily understood by the users. The meaning of the words (identifier in C) of this language has the same meaning as our daily life.

2) **Portable Language / Machine Independent Language:**

   Program written in C language can run in any computer with little or no modifications on the program. This nature is called portability. Hence C program can be written in any computer system.

3) **Easy to learn and understand:**

   Since C uses all English words, therefore it is easy to learn and understand the C program.

4) **Procedure oriented programming language:**

   This programming language only emphasize on the logic of the program used for solving the problem rather than the object.

5) **Modular/ Structured programming language:**

   In C programming a complex or long program can be divided into many sub – program which are called module or functions. Each module or function does the coherent work of the main program. Due to this the understanding of the program will be very easy and efficient.

6) **Small:**

   C is a language of few words, containing only a handful of terms, called keywords, which serve as the base on which the language's functionality is built.

7) **Structured Language**:

   - C allows programmer to divide program into modules
   - C provides all basic control structures
   - Use of subroutines that employ local variables

WHAT IS COMPILING AND COMPILER?

The process of converting or translating the program code written in high level language into the machine language is called compiling.

Source code: Program written in a particular programming language in its original form is called source code program.

Object code: It is the code produced by the compiler. It is same as or similar to machine language.

BASIC STRUCTURE OF THE C PROGRAM

Documentation section
Linkers section
Macro definition section / definition section
Global variable declaration section
Global function declaration section

main() function
{
  declaration part;
  execution part;
}

Function definition section

Documentation Section (question or the problem i.e. topic. This is not an executable statement. Remain as comment.)

Linkers Section (This line starts with #include which is called directive. It is used as a command to the compiler.)

Macro definition (We define the macro or constant or some short expression. For ex: # define PI 3.1416)

Global variable declaration (the variable which can use any function of the program is called global variables.)

Global function declaration (Those function or modules which can be called by any function of the program are called global functions.)

main() function (entry point of the program. Execution starts from this point.)

declaration part (we declare variable and function)

execution part (part where we write executable statement or instructions. Actually body of this program is execution part.)

PROGRAMMING TOOLS

- Algorithm, Flowchart, Pseudo code, Decision tree   *<detail refer class lecture>*

FUNDAMENTAL OF THE C PROGRAMMING

Header file:

Sentences that begin with a hash sign (#) are directives for the preprocessors. These directives are called header files. They are not executable part of the program but they should be included while writing any programs because they contain the definition or meaning of the library functions, keywords, words and pre – defined constants. The header file can be included by the following syntax:

#include<header file name> **Ex:** #include<stdio.h> **OR**

#include"header file name" **Ex:** #include"string.h"

The other header files which we will require for this level are listed below:

1. stdio.h	**5.** stdlib.h
2. conio.h	**6.** ctype.h
3. string.h	**7.** time.h
4. math.h	**8.** alloc.h etc.

Statements:

The sentence which is written in the program to do one work separately is called statement. Statements should be terminated by a semicolon (;) which is called the terminator. For ex: a = b =+ c; is a statement.

Braces {}:

The opening and closing braces are used in the beginning and end of the compound statement respectively. Every functions or the compound statement should contain this braces with appropriate form.

Compound Statement:

The statements written inside the braces which act as a single statement are called compound statement. Generally it consists of more than two statements.

For Ex:

```
{
a= b+c;
d=a+b;
}
```

this overall statement which is enclosed by braces is compound statement.

Input/output functions:

Those function which are used to send the data from the user or from the file are called input functions and functions used to display the result or output are called output functions. printf() and scanf() (commonly used output and input functions respectively)

Library functions:

Those functions whose definitions or job are pre - written in the library of the C program are called library functions. Printf (), scanf (), clrscr (), getch () etc. all are examples of library functions.

Comments:

The lines written within a program to make program user friendly are called comments. Comments are non-executable lines in the program. It helps to understand the logic of the program to solve the problem. Comment serves as documentation for the human reader of the program. Anything written between /*and */is called a comment.

Variable:

It is the name given to the memory Location to hold value. It is an entity. During the execution of the program value of the variable changes. For ex: int total, char name etc. here total and name are two variables.

Constants:

A value that doesn't change during the execution of the program is called constant. Constant is sometimes referred to as literals. Whatever the content is there in the variable which is fixed is called constant. It cannot be changed during the execution of the program.

An integer constant like 1234 is an int. a long constant is written with a terminal l (ell) or L, as in 123456789L; an integer constant too big to fit into an int will also be taken as long.

For Details,
Also refer String Constant, Character constant, floating point constant, numeric constant and so on.

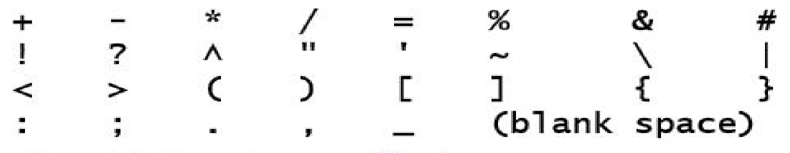
The C Character Sets

Set of characters that are used as building blocks to form basic program elements. The C character set consists of

– The 52 upper- and lower-case letters of the Latin alphabet

– The ten decimal digits

– Certain special characters



+	-	*	/	=	%	&	#
!	?	^	"	'	~	\	|
<	>	(	)	[	]	{	}
:	;	.	,	_	(blank space)		

Special characters used in C

Identifiers:

- Names given to various program elements such as variables, functions, labels, and other user defined items
- Naming **rule** for identifiers:
  - Can be a combination of letters, digits and underscore (_), in any order
  - The first character of an identifier must not be a digit.

Correct	**Incorrect**
X	"x"
y12	12y
nepal	nepal's
item_1	item 1
_temp	4th
order_no	order – no
welhams123	123welhams

- In an identifier, upper- and lowercase are treated as different.
  - For e.g., the identifier count is not equivalent to Count or COUNT.
- There is no restriction on the length of an identifier. However, only the first 31 characters are generally significant.
  - For e.g., if your C compiler recognizes only the first 3 characters, the identifiers pay and payment are same.

Keywords

- Keywords are reserved words that have standard, predefined meanings in C.
- You cannot use a keyword for any other purpose other than as a keyword in a C program. – For e.g., you cannot use a keyword for a variable name
- The ANSI C defines 32 keywords:

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

The printf() function:

This is one of the most common output functions used in C programming. Its main usage is to display the text, or value or both on the standard output device i.e. on the monitor. Depending upon what is to be displayed, the general syntax of printf () are as follows:

1. For displaying text only:

   printf("your text");
2. For displaying value only:

   printf("formal string/ type specifier", variable name);
3. For displaying both taxt and value:

   printf("text type specifier", variable name);

The function scanf ():

This is the most common data input function used in C. If we want to send the data to the computer then we use this function. It reads the data from the standard input device i.e. keyboard. General syntax of this function is: Scanf ("type specifier" , & variable_name) Here the symbol & is called address operator. For simplicity we can say that '&' represents the memory location of the variable and therefore called an address operator.

Scanf () / printf () format codes

Code	Use for	Code	Use for
%d	For integer constant	%h	For short integer
%c	For single character	%s	For string
%f	For floating point number	%l	For long
&ld	For long integer		

Other useful library functions:

Library function	Purpose
clrscr();	To clear the screen
getch();	To hold the output of the program unless any key is pressed
getche();	To read a character from a standard input device
strcpy();	To copy the string and assign
sqrt();	To find out the square root
abs();	To find the absolute value
strlen();	To find out the number of characters in a string
strcmp();	To compare two string
toupper();	To change a character in upper case
tolower();	To change a character into lower case

OPERATORS AND TYPES

Operators are nothing but the symbols which are used to derive certain operations. Operators are used in programs to manipulate and compute the data and variables. Therefore, operators are sometimes called as the **driver of the operation.** C contains large number of the operators.

Depending upon the **type of operator, operators are classified into** numbers of categories among them some important categories are as follows:

1) Binary Operator:
   a) Arithmetic Operators
   b) Relational Operators

c) Logical operators

d) Assignment Operators

2) Unary operators:

a) Increment and decrement Operators

b) Unary minus operator

3) Conditional Operator

1) Binary Operators:

The cord binary represents the meaning of this type of operator. Binary means ‘two’ therefore that type of operator which takes two operands to compute the operation is called binary operators.

a) Arithmetic Operators:

C includes all the fundamentals arithmetic operators. The symbol which is used to manipulate the various types of the arithmetic operation is called arithmetic operator.

Operators	**Meaning**	**Example**
+	Addition	a + b
-	Subtraction	a – b
*	Multiplication	a * b
/	Division	a / b
%	Modulo division / remainder operator	a % b

/* WAP which uses all the arithmetic operators. */

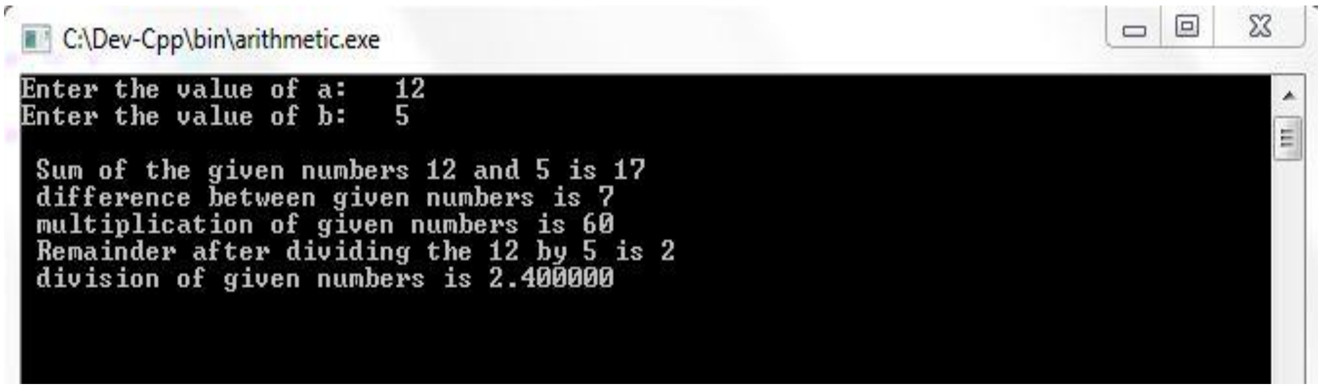
```
void main()
{
        int a, b, sum, sub, mul, mod;
        float div;
clrscr();
printf (“Enter the value of a: \t”);
        scanf (“%d” , &a);
printf (“Enter the value of b: \t”);
        scanf (“%d” , &b);
```

Explanation of the program

Here in the program we are written the statement div= (float) a/b; because here div is a variable of float type but a and b are variables of type int. While dividing int by int the result will be of type int i.e. it neglects the fractional part and prints only 2.000000 which is wrong result. This process is called type caste because here we change the caste of the result of the program first the result was int further we change it into float.

```
        sum=a+b;
        sub=a-b;
        mul=a*b;
        mod=a%b;
        div= (float) a/b;
printf ("\n Sum of the given numbers %d and %d is %d" , a, b, sum);
printf ("\n Difference between given numbers is %d" , sub);
printf ("\n Multiplication of given numbers is %d" , mul);
printf ("\n Remainder after dividing the %d by %d is %d" , a, b, mod);
printf ("\n Division of given numbers is %f" , div);
getch ();
}
```

Output



```
Enter the value of a:   12
Enter the value of b:   5

 Sum of the given numbers 12 and 5 is 17
 difference between given numbers is 7
 multiplication of given numbers is 60
 Remainder after dividing the 12 by 5 is 2
 division of given numbers is 2.400000
```

b) Relational Operators:

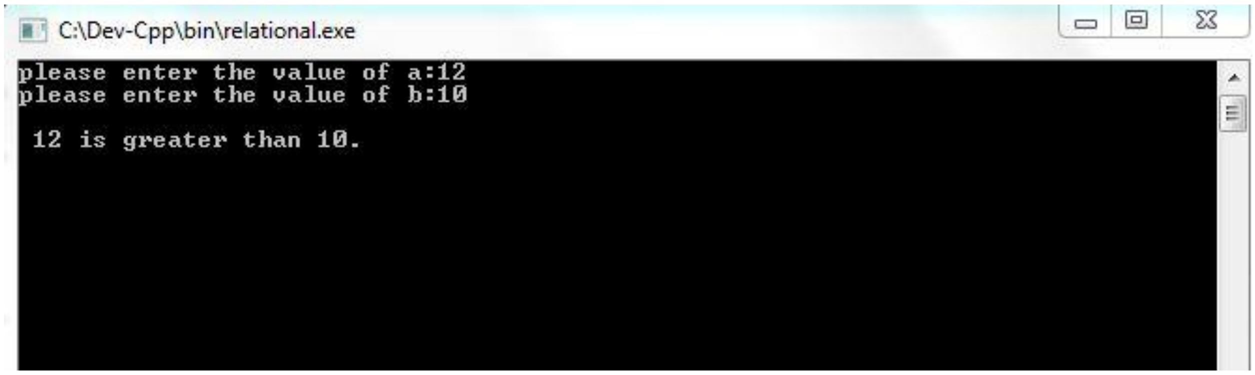
The symbols which are used to compare two expressions or two values or data are called relational operators. The result of the comparison is either true or false which is used to make a certain decisions. C supports six relational operators:

Operator	Meaning	Example
<	Is less than	X<Y
<=	Is less or equal	X<=Y
>	Is greater than	X>Y
>=	Is greater or equal to	X>=Y
==	Is equal to	X==Y
!=	Not equal to	X!=Y

/* WAP that finds out the greatest number between two given numbers */

```
main()   {
int a,b;
printf("please enter the value of a:");
scanf("%d",&a);
printf("please enter the value of b:");
scanf("%d",&b);
if(a>b)
        printf("\n %d is greater than %d.",a,b);
else
        printf("\n %d is greater than %d.",b,a);
getch();      }
```

Output:



C:\Dev-Cpp\bin\relational.exe

```
please enter the value of a:12
please enter the value of b:10

 12 is greater than 10.
```

c) Logical Operators:

If we have to combine two or more relational operations into single expressions then we use a logical operator which evaluates either true or false. Since it combines more than two relational expressions therefore it is also called compound relational operators. There are mainly three types of logical operators which are:

&&	meaning logical **AND**
||	meaning logical **OR**
!	meaning logical **NOT**

The AND and OR operators are widely used but NOT operator is rarely used. If 1 represents true and 0 represents false then truth table of these operators will be as:

X	Y	Z=X&&Y
0	0	0
0	1	0
1	0	0
1	1	1

Table: truth table of **AND** operator

X	Y	Z=X||Y
0	0	0
0	1	1
1	0	1
1	1	1

Table: truth table of **OR** operator

/* WAP that checks whether the number entered by the user is exactly divisible by 4 but not by 11. */

Hints: Before writing the code of the program we should know type of the operator used. For this, problem should be known very clearly. In this problem both the conditions mentioned (number should be divisible by 4 and number should not be divisible by 11) must be true. Therefore in this case we have to use logical AND operator.

```
main()
{
        int number;
        printf("Please enter a number:    ");
        scanf("%d",&number);
        if((number%4==0)&&(number%11!=0))
                printf("\n %d is exactly divisible by 4 but not by 11.");
        else
                printf("\n The condition is false.");
getch();
}
```

Output

```
Please enter a number:      12
12 is exactly divisible by 4 not by 11.
Please enter a number:       44
The condition is false.
```

Assignment

/* WAP which find out whether the given year is leap year or not.*/

Hints: *Before writing the code for this problem, we should know what type of year is called leap year or what is leap year? The year which is perfectly divisible by 4 but not by 100 or those year which is perfectly divisible by 400. Before writing the code for any problem we should know the problem i.e. definition of the problem, what the problem is and what is the logic that should be implemented to solve that problem? Only then we can write the proper program.*

d) Assignment operator:

The name of operator "assignment" says it is the operator which assigns something for a variable. Simple equal to sign (=) is called the assignment operator which assigns a value or expressions for a variable. It should take care that double equal sign (==) and assignment operator (=) are not same.

Example(s)

```
a=b+c;          b=a%b;
x=10;           y='a'; etc.
```

C Shorthand

C has a set of shorthand assignment operators. Shorthand assignment operator is the combination of two different arithmetic operators. The assignment of shorthand takes the form: V_{ao} =exp;

Where v represents variable, ao represents arithmetic operator and exp represents expressions. Here V_{ao} =exp; is equivalent to V = V_{ao} exp;

Shorthand	Equivalent meaning
A+=B;	A=A+B;
A-=B;	A=A-B;
A*=B;	A=A*B;
A/=B;	A=A/B;
A%=B;	A=A%B;

2) Unary Operator

Unary means one. Therefore that type of operator which takes only one operand is called unary operator. There are different types of unary operators among them some important are as follows:

a) Increment and decrement operator:

The operator which increases the value of a variable by unity is called increment operator and the operator which decrease the value of variable by unity is called decrement operator. Increment operator is ++ and decrement operator is --. The increment operator ++ adds 1 to its operand whereas decrement operator – subtracts 1 from its operand.

E.g. i++; which is equivalent to i=i+1; and i--; which is equivalent to i=i-1;

The unusual aspect is that ++ and – may be either be used either as **prefix** operators (before the variable, as in ++n), or **postfix** operators (after the variable: n++).In both cases, the effect is to increment the n. But the expression ++n increments n before its value is used, while n++ increments n after its value has been used. This means that in a context where the value is being used, not just the effect, ++n and n++ are different. If n is 5, then

X = n++;

Sets x to 5, but

X = ++n;

Sets x to 6. In both cases, n become 6.

Note: The increment and decrement operators can only be applied to variables; an expression like (i+j) ++ is illegal.

/* **WAP to illustrate the increment and decrement operators.***/

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a=5,b;
        b=a++;
printf("\n b=%d",b);
        b=++a;
printf("\n b=%d",b);
getch();      }
```

Explanation of the program:

In the program at first 5 is assigned to **a.** The statement **b=a++;** firstly assign the value of **a** and then increases the value of **a** by 1. Therefore 5 is stored on **b** and value of **a** will be 6 and **b=5** will be printed. But in the statement **b=++a;** firstly value of **a** is increased by 1 i.e. value of **a** will be 7 which further assigned to **b**. therefore value of **b** will be 7 and hence **b=7** will be printed.

Output

```
b=5
b=7
```

b) Unary minus sign:

This is simply minus sign which changes the value of variable with negative. In the binary minus i.e. in arithmetic minus sign it takes two operands which subtract the second operand from first one but in the case of unary it takes only one operand. For example: -n; is a statement.

3) Conditional Operator (ternary operator? :)

This is different type of operator. It is the combination of if – else statement as a condition operator. Therefore it is also called conditional statement. The statements

```
If(a>b)
        z=a;
else
        z=b;
```

Compute in z the greatest of and b. The conditional expression, written with the ternary operator "?:", provides an alternate way to write this and similar constructions. The general syntax of this type of expression is:

expr$_1$? expr$_2$:expr$_3$

Here the expression expr$_1$ (here which is conditions) is evaluated first. If it is non-zero (true), then the expression expr$_2$ is evaluated, and that is the value of the conditional expressions. Otherwise expr$_3$ is evaluated, and that is the value. Only one expr$_2$ and expr$_3$ is evaluated not both at a time. Thus to set z to the maximum of a and b,

z = (a > b) ? a : b;

The conditional operator takes three operands; therefore we can say that it is a ternary operator.

/* WAP that finds out the greatest among two numbers using conditional operator.*/

```
 #include<stdio.h>
#include<conio.h>
Void main()
{
        int a,b;
        int greater;
        printf("\n please enter any two numbers:");
        scanf("%d %d",&a,&b);
        (a>b) ? greater=a : greater=b;
        printf("\n\n greater number is %d",greater);
getch();        }
```

Explanation of the program:
In this program if **a** is greater than **b** then it assigns **a** as greater otherwise **b** is assigned to greater. And finally greater in which is greatest number is assigned is printed.

Among these operators which are mentioned there are other types of operators are also available in C program such as bitwise operator, Special operators (sizeof operator, pointer operator, comma operator, member selection operator etc.). But important operators are explained.

Precedence and Associativity of operators in C programming

Each operator in C has its own precedence associated with it. The Precedence means in the expression containing more than one operator, which one is evaluated first and then after. The operators at higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from left or from right depending from the level. This is known as Associativity of an operator.

Precedence of Arithmetic operators and unary operators:

Operators	**Associativity**	**Precedence**
++, --	Right to Left	Highest
- (unary minus)	Left to right	
*, /, %	Left to right	
+, -	Left to right	
!	Right to left	
>, >=, <, <=	Left to right	
==, !=	Left to right	
&&	Left to right	
||	Left to right	Lowest

ASSIGNMENT – II

A. Write the algorithm and flowchart then program that checks whether the number entered by the user is divisible by 3 but not by 7.
B. WAP to check whether the entered year is leap year or not.
C. WAP that converts binary number into the equivalent decimal value.
   [Hints: bin=bin/10 dec+=dig*pow(2,i);]

Data input and output:

The important of C programming is its ability to fascinate the user as far as possible. To do this we should able to handle I/O. All the I/O operations are carried out through function call such as printf( ), scanf( ) etc.

These types of functions which are used to read and display the data are collectively known as standard I/O library functions. Each program that uses these types of functions should include the header file <stdio.h> which stands for standard input output header file. Hence the printf() function moves the data from computer memory to standard output device i.e. monitor whereas scanf() functions enters the data from standard input device i.e. keyboard and stores it in computer main memory.

Data Input functions	**Data output functions**
getch()	putchar()
getche()	putc()
getchar()	puts()
gets()	printf()

Escape sequence characters

Those characters which escape the sequence of the data are called escape sequence characters. Some examples are:

Character	**Function**	**Character**	**Function**
\a	alert (bell)	\\	backslash
\b	Backspace	\?	To print question mark
\f	Form feed	\’	To print single quote
\n	New line	\”	To print double quote
\r	Carriage return	\ooo	Octal number
\t	Horizontal tab	\xhh	Hexadecimal number
\v	Vertical tab		

Use of getchar() and putchar() functions:

Both functions are character holding functions that means they can process only one character at a time. By the name of function getchar(), it is used to read a character from a standard input device i.e. keyboard. Similarly the function putchar() is used to put or print a character on the monitor. The general syntax of these functions is:

variable = getchar();

putchar (variable);

/* WAP which reads a character from the keyboard and print in upper case.*/

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
Void main()
{
        char ch;
        printf("Please enter a character:");
        ch=getchar( );      /* reading a character.*/
        ch=toupper(ch); /* changing the character into a upper case.*/
        putchar(ch);  /*printing the character.*/
getch();              }
```

Explanation of the program:

We can write in place of putchar (ch); the statement printf ("%c",ch); both result same output.

Refer comments

Use of gets() and puts() functions:

As in the case of getchar() and putchar() where 'char' represents character. In the same way here in gets() and puts(), 's' represents string. String is set of character. These functions are string functions which are used in manipulation of strings i.e. reading and writing the string.

Control Statement

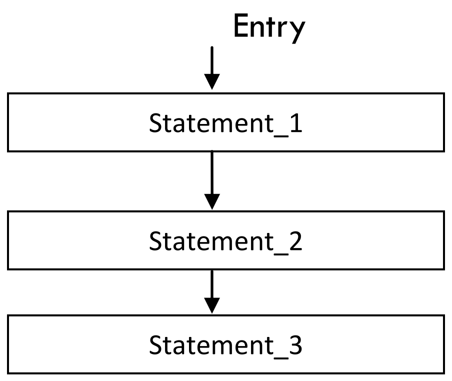
C program consists of set of instructions which are executed in sequentially order as written in any high level programming language. But many cases may arise where we have to choose the statement to be executed. At this instant we require the instruction which transfer the control of the program from one point to another point in the program and sometimes we have to repeat the same task for the number of times. C has this type of facility. This type of statement or instruction which transfers the control of the execution of the program from one point to another point is called control statement. ***C supports following types of control statements:***

1) **Sequential Control Statement**
2) **Decision making control Statement**
    a) simple if Statement
    b) if – else Statement
    c) if – else –if Statement
    d) nested if Statement
    e) conditional operator Statement
    f) switch Statement
3) **Iteration or looping Statement**
    a) while loop
    b) do – while loop
    c) for loop
4) **Special control statement**
    a) goto statement
    b) break statement
    c) continue statement

1) Sequential Control Statement:

If the flow of the statements in sequential order that means as written in the source code. The sequential control statement ensures that the instructions in the program are executed in the same order as they appeared in the program.

Flow chart of sequential control statement is:



2) Decision making control Statement:

Those statements which are used to make some decisions are called decision making statements. These statements are also called as selection statement. Using these statements we can select the required statement to be executed. In this type of structures the test condition decides the flow of the execution.

a) Simple if statement:

The general syntax of this function is:

```
if(condition)          // Here if should be in lower case
{
true statement (s);
}
next statement;
```

Explanation of the syntax:

Here, initially condition (which is logical condition) is checked. If the condition is true then **true statement** enclosed by a pair of braces is executed and then **next statement** will be executed otherwise (i.e. if the condition is false then).

/*WAP to check whether the number entered by the user is even or odd.*/

Problem analysis:

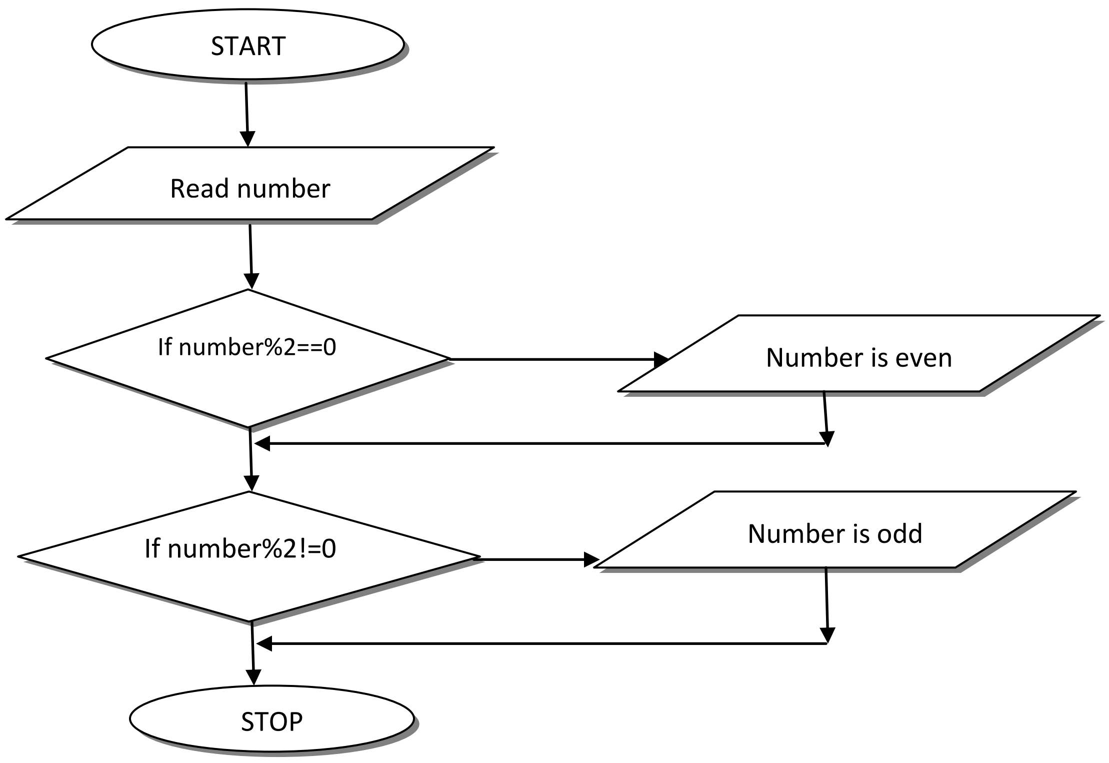
Before writing the code for this program we should know what type of program is called even number and what type of number is called odd number. The number which is perfectly divisible by 2 is called even number and remaining other are called odd numbers.

Algorithm of the program:

Steps:

I. Start the program
II. Read a number
III. If the number is perfectly divisible by 2 then print "the number is even."
IV. If the number is not divisible by 2 then print "the number is odd."
V. Stop the program

Flowchart of the program:



Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
        {
        int number;
        printf("enter a number:");
        scanf("%d",&number);
        if(number%2==0)
        printf("\n entered number is even");
        if(number%2!=0)
        printf("\n entered number is odd");
getch();
}
```

Output:

```
enter a number: 12
entered number is even.
enter a number: 3
entered number is odd.
```

Conclusion:

Hence, program to check whether the number entered by the user is even or odd is done.

b) if – else statement:

"if-else" statement is the real decision making statement. In simple if statement there is no alternative i.e. if the condition is false then there is no any alternative but in case of if else statement, there is alternative. Here, else is keyword. The general syntax of this statement is:

```
        if(condition)
        {
                true_statement(s)
        }
        else
        {
                false_statement(s)
        }
        next_statement;
```

Explanation of the syntax:

Here, firstly the condition is checked. If the condition is true then the true statement(s) will be executed as in the simple if statement otherwise false statement(s) will be executed. That means either true or false, only one statement will execute not both at a time. After executing one of the statements, then next statement will be executed.

/* WAP to find whether the entered number is odd or even.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int number;
```

Output:

```
please enter a number: 13
entered number is odd.

please enter a number: 18
entered number is even.
```

```
        printf("please enter a number:");
        scanf("%d",&number);
        if(number%2==0)
                printf("\n entered number is even.");
        else
                printf("\n entered number is odd.");
getch();    }
```

c) if – else – if statement (if – else ladder):

In this form of if statement, conditions are checked first. If the condition is found to be true then the statements associated with the condition will be executed and other conditions will be skipped. If none of the conditions are true, then it will execute the statement of else side. It takes the **following form of the syntax:**

```
        if(condition1)
                {
        statement1;
                }
        else if (condition2)
                {
        statement2;
                }
        else if (condition3)
                {
        statement3;
                }
        else……….
        …
        ..
        else if(condition_n)
                {
        statement_n
                }
        else
                {
        statement_n+1;
                }
        next_statement;
```

/*WAP that takes average marks obtained by a student. Your program should display the result and division of that student.*/

[if average mark >=80, distinction, if average marks >=60 but <80, first division, if average marks >=45 but <60, second division, if average marks >=35 but <45, third division. To pass, the average marks should be greater or equal to 35.]

```
#include<stdio.h>
#include<conio.h>
void main()
{
        float ave;
        printf("enter the average marks obtained by the student: \t");
        scanf("%f",&ave);
        if(ave>=80)
        printf("\n  result=pass.\n division=distinction.");
        else if(ave<80 && ave>=60)
        printf("\n  result=pass.\n division=first division.");
        else if(ave<60 && ave>=45)
        printf("\n  result=pass.\n division=second division.");
        else if(ave<45 && ave>=35)
        printf("\n  result=pass.\n division=third division.");
        else
        printf("\n  result=fail.\n division=no division.");
getch();        }
```

d) Nested if statement:

If an "if" statement contains if statement within the statement then such type of condition is called nested if condition. Nested if is an if that has another if in its if's body or in its else's body. In other word, "if else" statement within another "if-else" body is known as nested if. The general syntax of this type of statement is:

```
if (condition)
{
        if (condition)
                {
                        statement(s);
                }
        else
                {
                        statement(s);
                }
}
else
{
statement(s);           }
```

/* WAP that reads numbers a, b, c and prints the largest number among them using nested if structure.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,c;
        printf("please enter any three integer numbers:" \t);
        scanf("%d %d %d",&a,&b,&c);
        if(a>b)
                {
                if(a>c)
                        printf("\n %d is largest",a);
                else
                        printf("\n %d is largest",c);
                }
        else if(b>c)
                printf("\n %d is largest",b);
        else
```

```
                printf("\n %d is largest",c);
getch();        }
```

e) Conditional operator statement:

The name of the statement type suggests that it is that type of statement which uses conditional operator for the operation. Simply this is the combination of simple if else statement. Syntax of this type of statement is:

```
        expression1 ? expression2 : expression3;
```

In this if the expression1 is true then expression2 is evaluated otherwise expression3 will be evaluated. This is equivalent to:

```
If(expression1)
expression2;
else
expression3;
```

/* WAP that finds out the greater number among two numbers using conditional operator statement.*/

```
#include<stdio.h>
#include<conio.h>
main()
        {
                int a,b;
                int greater;
                printf("please enter any two numbers: ");
                scanf("%d %d",&a,&b);
                (a>b) ? greater = a : greater = b;
                printf("\n\n greater number is %d",greater);
getch();        }
```

f) Switch statement:

if we have to create a program of a menu system then we can use switch statement. This is similar to if-else-if statement. Use of this statement makes the program more user friendly and more standard. The general syntax for this statement is:

```
switch(choice)
{
case 1:
        statement(s);
        break;
case 2:
        statement(s);
        break;
…..
…..
default:
        statement(s);
        break;
}
```

Explanation of the syntax:

If the choice is 1, then only executes the statements written in case 1 block, then it directly transfers the control of the program out of the switch statement. If the choice is not matched then it executes the statements written in default block and transfers the control out of statement.

Note

If the choice is of character type i.e. alphabet then in case of writing case 1, case 2, …….we have to write case 'a', case 'b', etc.

/*Write a menu type program that reads two integer's value and ask for the choice. If you press 1 then your program should add them and display, and similarly on pressing 2 subtraction, on pressing 3 multiplication, on pressing 4 division and on pressing 5 remainder should display. If any other than 1-5 is pressed then your program should display an appropriate message.*/

```
        #include<stdio.h>
        #include<conio.h>
        void main()
        {
        int a, b, res, choice;
        float div;
printf("please enter any two integer numbers:        ");
scanf("%d %d", &a, &b);
printf("\n 1. Addition \n 2. Subtraction \n 3. Multiplication\n 4.Division \n
5.Remainder");
printf("\n \n please enter your choice(1-5):\t");
scanf("%d",&choice);
switch(choice)
{
case 1:
        res = a+b;
        printf("\n addition of given two numbers is %d",res);
        break;
case 2:
        res = a-b;
        printf("\n subtraction of given two numbers is %d",res);
        break;
case 3:
        res = a*b;
        printf("\n Multipliction of given two numbers is %d",res);
        break;
case 4:
        div = (float)a/b;    // type casting
        printf("\n Division of given two numbers is %d",div);
        break;
case 5:
        res = a%b;
        printf("\n Remainder after dividing %d by %d is %d",a,b,res);
        break;
default:
        printf("\n your choice is out of range. Please enter again.");
        break;
}
getch();        }
```

Output of the program

```
please enter any two integer numbers: 12  4
    1. addition
    2. subtraction
    3. multiplication
    4. division
    5. remainder
    please enter your choice (1-5):      3
    multiplication of given numbers is 48

please enter any two integer numbers: 12  4
    1. addition
    2. subtraction
    3. multiplication
    4. division
    5. remainder
    please enter your choice (1-5):      7
    your choice is out of range. Please enter again
```

3. Iteration or Looping Statement

If we have to do same task for several times then use of sequential programming approach becomes more difficult, time consuming and less user friendly. In this situation we need a type of statement which can perform the same task for the specified times. This type of statement is called looping statement and the process is called iteration. In general this type of statement includes four things:

- Setting or initializing a condition variable.
- Execution of the statement inside the loop.
- Updating the condition variable.
- Test the specified value of the condition variable.

Here the condition may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been reached.

There are mainly 3 types of looping statement used in C and they are:

a. while loop/statement
b. do while statement
c. for statement

a. The while statement:

This is a looping statement where the statements that are written inside the loop executed till the condition is true. The general syntax of this looping statement is

```
initialization;
while(test_expression/condition)
{
statement(s); // body of the loop
expression;
}
next_statement;
```

The "while statement" is an entry-controlled loop statement. At first the test condition is evaluated and if the condition is true, then the body of the loop is executed. After the execution of the body, the test condition is again tested and again it yields true then again the loop will be executed. This process will go on continuously until the test condition yields result as false. When the test condition become false then the control of the program will transfer just outside the loop and then next_statement will be executed.

The body of the loop may or may not contain more than one statement. The pair of braces will only needed if the body of the loop contain two or more than two statements i.e. if the body of the loop is compound statement. But for good practice it is better to put pair of braces even for a single statement.

/* Using while statement, WAP to print the numbers from 0 to 100.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i;
        printf("the required series given below:\n");
        i=0;                    // initialization
        while(i<=100)                   // test condition
                {
        printf("%d \t" , i);  // body of the loop
        i++;                    // expression
                }
        getch();        }
```

b. do while loop:

"do" means perform the work or execute the statements and **"while"** means until the condition is satisfied. That means in "do-while" loop statement, statements are executed first and then condition is checked. Therefore this statement is exit controlled statement because the condition for the loop is evaluated after the execution of the statements. The general syntax for this looping statement is:

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

In the **syntax** of do-while statement it should be take care that, the while statement should end with semicolon. This is also the difference between while loop and do-while loop.

Differences between the while loop and do-while loop statement:

S.N	While loop	S.N	Do-while loop
1.	It is an entry controlled loop. That means the entry point decides whether to enter into the loop or not.	1.	It is exit controlled loop. That means the exit point of the loop decides whether to again enter into the loop or not.
2.	The test condition is tested outside the loop before entering into the loop.	2.	The test condition is tested within the loop before exiting the loop.
3.	If the condition is not satisfied then the statement inside the loop will not be executed.	3.	Even the condition is not satisfied at least one time the loop will be executed.

4.	**Syntax:**
	while(test expression)
	{
	body of the loop
	}
		4.	**Syntax:**
			do
			{
			body of the loop
			} while(test expression);

/* WAP to read a number from keyboard until a zero or negative number is keyed in. Finally, calculate sum and average of entered numbers.*/

```
void main ()
{
        int num, count=0;
        float sum=0,avg;
        clrscr();
        do
           {
           printf("\n Enter number:\t");
           scanf("%d",&num);
           sum+=num;
           count++;
           } while (num>0);
           sum=sum-num;
           avg=(sum)/(count-1);
           printf("\n The sum is:\t %d ",sum);
           printf("\n The average is:\t %f ",avg);
           getch();        }
```

/* here the last number either 0 or negative number is not included in average. Thus, the final number entered is excluded from the sum and hence sum-num and count-1.*/

Output:

```
      Enter number:          10
      Enter number:          20
      Enter number:          15
      Enter number:          5
      Enter number:          0
      The sum is:        50.000000
      The average is:    12.500000
```

C. The for loop/statement:

The for loop is useful to execute a statement for a number of times. When the number of repetitions is known in advance, the use of this loop will be more efficient. Thus, this loop is also known as a determinate or definite loop.

***Syntax*:**

```
        for (counter initialization; test condition; increment or decrement)
        {
        Statement(s); or body of loop
        }
```

/* WAP that asks an integer number n and calculate sum of all natural number from 1 to n.*/

```
void main ()
  {
  int num, i, sum=0;
  clrscr();
  printf("\n Enter a number n: \t");
  scanf("%d", &num);
   for (i=1; i<=num; i++);
       sum+=i;
  printf("\n The sum is: %d", sum);
  getch();     }
```

Output:

```
Enter a number n:      10
The sum is 55.
```

4) Special control statement / Jumping statement:

These statements are used to jump the statements as our requirement. Therefore, this is called jumping statement. There are mainly 3 types of jumping statements. They are:

a) The goto statement:

This statement is used to transfer the control of the program anywhere within the program. Using these statements we can break the sequence of execution of the program without having any test conditions and transfer the control to the statement having the statement label rather than any other statements.

Two types of goto statement.

- forward goto statement
- backward goto statement

Syntax of forward goto statement	Syntax of backward goto statement
goto label: ⟶ …………….. …………….. …………….. …………….. …………….. *label:* ⟵ ……………..	*label:* ⟵ …………….. …………….. …………….. …………….. …………….. *goto label:* ⟶ ……………..

While programming it is better not to use goto statement because use of this statement makes the program poor. The use of goto statement in the program is considered as poor programming. Use of this statement is accepted only statement.

/* WAP that generates the multiplication table of a number entered by the user without using any looping structure. (Use goto statement).*/

```
#include<stdio.h>
#include<conio.h>
void main ()
{
        int i=1; num; mul;
        printf("\n please enter the number:");
        scanf("%d",&num);
        up:
        mul=i*num;
        i++
        printf("\n %d*%d=%d",num,I,mul);
        if(i<=10)
        goto up;
        getch();
}
```

b) The break statement:

The *break* statement terminates the execution of the loop and the control is transferred to the statement immediately following the loop. Generally, the loop is terminated when its test condition becomes false. But if we have to terminate loop instantly without testing loop termination condition, the break statement is useful. The syntax for this is:

```
        break;
```

break statement is also used I switch statement which causes a transfer of control out of the entire switch statement, to the first statement following the switch statement.

/* What is the output of the following program.*/

```
void main()
{
        int i;
        clrscr();
        for (i=1; i<10; i++)
                {
                printf("\t %d",i);
                if (i==5)
                        break;
                }
        getch();
}
```

Explanation of the program:

The counter variable i is initialized to 1 and the test condition is i<10. Thus, the loop body is to be executed 9 times (i.e. 1 to 9). But the use of break statement within the body of the loop has terminated the loop when the value of I becomes 5. Thus, its output is only numbers 1 to 5 instead 1 to 9.

Output:

```
1       2       3       4       5
```

C. The continue statement:

This is also another jumping instruction. The "continue" statement and the "break" statement are seemed to be similar but there is huge difference between them. The "continue" statement is used to escape the certain statement if some specified condition is satisfied and again return to the loop. The "break" statements stops the loop if the condition for the break statement is true but in the case of continue statement loop will not be stopped. This is the major difference between these two statements.

The general syntax for "continue" statement is:

```
Loop statement
{
statement_1;
...............
...............
if (condition2)
continue;
...............
...............
```

Explanation of the syntax:

When the execution of the program reach the loop firstly condition1 is checked, if the condition1 is true only then it enters into the loop. After entering into the loop it executes the statement_1 block and checks the condition2. If the condition2 is true then it executes the continue statement and transfers the control directly to the loop without executing the remaining statements after continue statement. And again loop will be continued. In this type of looping where only continue statement is used the number is execution time of loop depends only on the loop condition i.e. condition1.

```
        statement_2;
        }
        next_statement;
```

/* WAP that prints the number from 1 to 15 except 7 and 11.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i;
        for(i=1; i<=15; i++)
                {
                if (i==7 || i==11)
                continue;
                else
                printf("%d \t",i);
                }
getch();        }
```

There are also other jump instructions such as **return** and **exit.** The detail on return statement will be studied in function and exit statement is used to terminate the whole program at our required point.

===

UP TO HERE SOLVED EXAMPLES

===

1. /* WAP to print sum, product and quotient of two numbers entered by the user.*/

2. /*WAP to compute the area of sphere by reading the radius of sphere.*/

```
        #include<stdio.h>
        #include<conio.h>
        #define PI 3.14159
        void main()
        {
```

```
        float radius;
        float area;
        printf("\n enter the radius of the sphere:      ");
        scanf("%f",&radius);
        area=4*PI*radius*radius;
        printf("\n\n Area of sphere having radius %f is %f.",radius,area);
        getch();
        }
```

3. /* WAP that reverses and sums up the digits of an integer.*/

```
#include<stdio.h>
#include<conio.h>
main()
{
    int n,m;
    int sum=0;
    int rev=0;
    printf("Enter any digit number");
    scanf("%d",&n);
    while(n>0)
    {
            m=n%10;
            sum+=m;
            rev=rev*10+m;
            n=n/10;
    }
    printf("The sum of the no is %d",sum);
    printf("The reverse of the number is %d",rev);
getch();
    }
```

4. /* WAP using nested loop to print the multiplication table for even numbers from 2 to 10.*/

```
        void main()
        {
        int i, j, mul;
        for(j=1; j<=10; j++)
        {
                for(i=2; i<=10; i= i+2)
                {
                mul=i*j;
                printf("%d*%d=%d \t",i,j,mul);
                }
printf("\n");
}
        getch();
}
```

5. /* WAP to find factorial of a supplied number.*/

```
#include<stdio.h>
        #include<conio.h>
        void main()
        {
        int num, i;
        long int fact=1;
        printf("\n Please enter a number whose factorial is to be determined:\t");
        scanf("%d",&num);
        printf("\n\n\n factorial of %d is:\n");
        for(i=1; i<=num; i++)
                {
                fact*=i;
                if(i<num)
                printf("%d*",i);
                else
                printf("%d",i);
                }
        printf("=%ld",fact);
        getch();
        }
```

Sample output of the program:

```
Please enter a number whose factorial is to be determined: 6
Factorial of 1326 is:
1*2*3*4*5*6=720
```

6. /* WAP to display alphabets from a to z.*/

7. /*WAP to read a character from keyboard and convert it into uppercase if it is in lowercase and vice-versa.*/

8. /*WAP to read an integer number n from keyboard and display message "Welcome to our college" n times.*/

9. /* WAP to read a number from keyboard and check it for palindrome.*/

==

EXERCISE

==

1. Discuss historical development of C language.
2. What are the main features of C language?
3. What are the general characteristics of C?
4. Discuss importance of C language.
5. What is the basic structure of C program? Explain each section of the structure.
6. What do we mean by comment? How can comments included within a C program?
7. What is a source code or a program? Why is compilation needed before executing a C program?
8. What do you mean by control statements in C?
9. What is type casting? Explain with example.
10. What is operator precedence and Associativity? Explain with example.
11. What is an operator in C language? Discuss the different types of operators used in C language.
12. Explain syntax, working and uses of for loop in C.
13. What is branching? What is the purpose of the if-else statement?
14. What is meant by loop? Why is looping necessary in programming? Explain with example.
15. Distinguish between the following:
    a. do-while and while loop
    b. break and continue statement
    c. finite and infinite loop
    d. Branching (Decision making statement) and looping.

FUNCTIONS

Divide and Conquer

– Construct a program from smaller pieces or components

– These smaller pieces are called modules

– Each piece is more manageable than the original program

A function is defined as a self-contained block of statements that performs a particular task or job. This is a logical unit composed of a number of statements grouped into a single unit.

Classified into two categories namely library functions and user defined functions

S.N	Library Functions	S.N	User defined functions
1.	The library functions are already predefined and stored on the system library.	1.	The user defined functions are created by the user while writing the program.
2.	In the library functions, the name of the functions are fixed i.e. we cannot change the function name as per our requirement.	2.	In the user defined function, we can give name for the functions as our requirement except the keywords and library functions name.
3.	Ex: printf(), scanf() etc.	3.	Ex: factorial () etc.

ADVANTAGES OF FUNCTIONS

- **Manageability:**
  Using functions for specific job, it is easier to write programs and keep track of what they are doing. It makes programs significantly easier to understand and maintain by breaking them up into easily manageable chunks.
- **Code Reusability**
  A single function can be used multiple times by a single program from different places or different programs (i.e. the same code is used again and again reused.). The C standard library is an example of functions being reused.
- Debugging, testing and maintaining the program becomes much easier.

- A function can be reused i.e. new programs can be developed with the help of these codes.
- Use of functions increases the modularity of the program.
- Readability of the program increases i.e. programs becomes easy to read.
- This makes the execution work fast.
- Easy to divide the work to many programmers.

/* WAP to compute combination of numbers nCr without using functions.*/

```
void main()
 {
Long f1=1, f2=1, f3=1, comb;
Int n, c, r, i;
clrscr();
printf("enter n and r!! \t");
scanf("%d %d",&n, &r);
for(i=1; i<=n; i++);
         f1*=i;         /* To calculate n! */
for(i=1; i<=(n-r); i++)
         f2*=i;         /* To calculate n-r! */
for(i=1; i<=r; i++);
         f3*=i;         /* To calculate r! */
comb=f1/(f2*f3);
printf("\n The combination is %ld",comb);
getch();
}
```

Explanation of the program

Here, to compute combination i.e. ^nC_r = n!/((n-r)!r!). We have to calculate factorial of n, n-r and r. The logic and code is similar in each case. We are writing a code to calculate factorial of a number three times, one for n!, the second one for n-r! and the third for r!. Thus, this program has repeated code which is not good anf efficient. We can make a function to calculate factorial of a number and call this as required multiple of times instead of this.

Output:

```
enter n and r!! 6    2
The combination is 15


Enter n and r!! 5    3
The combination is   10
```

/* WAP to calculate combination of numbers nCr using function.*/

```
long factorial (int n)
{
long fact=1;
```

In this example, a function called factorial () is defined once and it is called three times to calculate the factorial of n, n-r and r.

```
int i;
for (i=1; i<=n; i++)
        fact*=i;
return fact;
}
void main ()
{
long f1=1, f2=1, f3=1, comb;
int n, c, r;
clrscr();
printf ("enter n and r!! \t");
scanf("%d %d", &n, &r);
        f1=factorial(n);     /*call function factorial() to calculate n! */
        f2=factorial(n-r);   /*call again factorial() to calculate n-r! */
        f3=factorial(r);             /*call again factorial() to calculate r! */
        comb=f1/(f2*f3);
        printf("\n The combination is %ld", comb);
getch();
}
```

Output:

```
enter n and r!! 6    2
The combination is 15
```

Why Use Functions?

Why write separate functions at all? Why not squeeze the entire logic into one function, main ( )?

Two reasons:

a) Writing functions avoids rewriting the same code over and over. Suppose you have a section of code in your program that calculates area of triangle. If later in the program you want to calculate area of a different triangle, you won't like it if you are required to write the same instructions all over again. Instead, you would prefer to jump to a 'section of code' that calculates area and then jump back to the place from where you left off. This section of code is nothing but a function.

b) Makes program easier to design and understand: If the operation of the program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently.

Structure of a function:

```
function_name (argument list);
argument declaration;
{
local variable declaration;
statements;
return (expression);
}
```

Function prototype:

All the functions must be declared before they used in the program. The prototype of the function tells in advance about the characteristics of the function used in the program. From the function prototype we can know the type of the arguments that takes the function and return the type of the function (i.e. the type data that the function returns). The function prototype is of form:

```
return_type    function_name  (type arg1, type arg2 … type arg n);
```

Here return_type indicates the data type to the variable returned by the function; function_name indicates the name of the function; type indicates the type of the argument and arg1, arg2 etc. indicates the arguments that are passed to the function.

Function definition:

It is the actual body of the function. In fact it starts with the function header which is similar to the function prototype but it doesn't end with a semicolon. It defines the type of the value that the function returns, its parameters and the statements that are executed when the function is called.

Its general form is as follows:

```
return_type  function_name (type arg1, type arg2 …….type arg n)
{
statements;
return (expression);
}
```

We can write the code for the function definition after or before the main () function. If we have written the function definition before the main () function then there is no need of function prototype OR function definition. But if we have written after the main (), then we should declare the function before main ().

Function Call:

A function can simply be called by using the name of the function in the statement and by supplying the arguments. The function in which the function call statement is contained is called **calling function** and the function which is called by the calling function is called the **called function.**

Formal argument and Actual Argument:

Those argument contained by the called function is called actual argument. This means the arguments we supply while calling a function is actual argument. Those argument contained in the function definition is called the formal arguments.

/* **Define a function** greater () **to find the greatest number among two numbers. WAP that uses this function to find the greatest number among three numbers.***/

```
#include<stdio.h>
#include<conio.h>
int greater(int x, int y)               [6----10]
{
   if(x>y)                              [7------11]
         return(x);                     [12]
   else
         return(y);                     [8]
   }
main()                                  [1]
{
     int a, b, c, d, e;
     clrscr();                          [2]
```

```
    printf("Enter three numbers");              [3]
    scanf("%d %d %d", &a, &b, &c);              [4]
/* let a=34, b=56 and c=10 */
    d=greater(a,b);                             [5]
    e=greater(d,c);                             [9]
    printf("The greater number is: %d",e);      [13]
    getch();                                    [14]
    }
```

Here, the execution of the program starts from the function main().

Category of functions according to return value and arguments:

According to the arguments and return values present in functions, we can categorize the functions in three categories:

Category 1: Functions with no arguments and no return values.

Category 2: Functions with arguments and no return values.

Category 3: Functions with arguments and return values.

1) **Functions with no arguments and no return values:**

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it doesn't return a value, the calling function does not receive any data from the called function. Thus, in such type of functions, there is no data transfer between the calling function and the called function. This type of function is defined as:

```
void function_name()
    {
/* body of the function */
}
```

The keyword void means the function doesn't return any value. There is no argument within the parenthesis which implies function has no argument and it does not receive any data from the called function.

/* WAP to illustrate the "FUNCTIONS WITH NO ARGUMENTS AND NO RETURN VALUES". */

```
void addition()
{
int a, b, sum;
printf("\n Enter two numbers:\t");
scanf("%d %d", &a, &b);
sum = a+b;
printf("\n The sum is %d",sum);
}
void main()
{
clrscr();
addition();
getch();
}
```

Explanation of the program

In this example, the function **addition()** has neither arguments nor return values. There is no data transfer between function **addition()** and **main().** The function itself reads data from the keyboard and display in the screen. As it does not return any value, there is no return statement within the function.

Output:

```
Enter two numbers: 45        90
The sum is   135
```

2) Functions with arguments and no return values.

The function of this category has arguments and receives the data from the calling function. The function completes the task and does not return any values to the calling function. Such types of functions are defined as:

```
Void function_name(argument list)
        {
        Body of the function
        }
```

/* WAP to illustrate "FUNCTIONS WITH ARGUMENTS BUT NO RETURN VALUES". */

```
void addition (int a, int b)
{
int sum;
sum=a+b;
printf("\n The sum is %d",sum);
}
void main()
{
int a, b;
clrscr();
printf("\n Enter two numbers:\t");
scanf("%d %d",&a, &b);
addition(a,b);
getch();        }
```

Explanation of the program:

In this example, the function **addition()** does not read data from the keyboard directly. It receives the data from the **main()** function i.e. calling function. It adds these two numbers and displays the sum in the screen but it does not return any value to the calling function. Thus, there is no return statement within this function.

Output:

```
Enter two numbers:     50      40
The sum is       90
```

3) Functions with arguments and return values.

The function of this category has arguments and receives the data from the calling function. After completing its task, it returns the result to the calling function through return statement. Thus, there is data transfer between called function and calling function using return values and arguments. These type of functions are defined as:

```
return_type  function_name (argument list)
{
body of the function
}
```

/* WAP to illustrate FUNCTIONS WITH ARGUMENTS AND RETURN VALUES. */

```
int addition (int a, int b)
{
int sum;
sum=a+b;
return sum;
}
void main()
{
int a, b, sum;
clrscr();
printf("\n Enter two numbers:\t");
scanf("%d %d",&a, &b);
sum = addition(a,b);
printf("\n The sum is \t %d",sum);\
getch();        }
```

Explanation of the program:

In this example, the function **addition()** has return type *int* and it has two arguments of type *int.* It receives two numbers from the calling function and adds them. The sum is returned to the calling function i.e. **main()** which assigns the returned value to some variable and display to the screen.

Output:

```
Enter two numbers:     78      56
The sum is:      134
```

/* WAP to generate the Fibonacci series.*/

```
#include<stdio.h>
#include<conio.h>
main()
{
int n1=0,n2=1,temp,i,n;
printf("Enter the numbers of terms:");
scanf("%d",&n);
printf("%d\t%d",n1,n2);
for(i=3;i<=n;i++)
{
temp=n1+n2;
printf("%d\t",temp);
n1=n2;
n2=temp;
}
getch();
}
```

Explanation of the Fibonacci series

In Fibonacci series, the first two terms are 0 and 1. Then, sum of 0 and 1 i.e. 1 will be third term. The sum of 1(second term) and 1(third term) will be fourth term i.e. 2. Then the sum of 1 and 2 will be fifth term and so on up to n terms. Logic: *if number of term is 1 i.e. 1 first term, return 0, if number of term is 1 i.e. first term return the sum of previous two terms.*

Output:

```
Enter the number of terms:     10
0     1     1     2     3     5     8     13     21     34
```

Difference between Recursion and Iteration:

S.N	Recursion	S.N	Iteration
1.	A function is called from the definition of the same function to de repeated task.	1.	Loop is used to do repeated task.
2.	Recursion is a top-down approach to problem solving. It divides the problem in to pieces.	2.	Iteration is like bottom-up approach; it begins with what is known and from this it constructs the solution step by step.
3.	In recursion a function calls to itself until some condition will be satisfied.	3.	In iteration, a function does not call to itself.
4.	Problem could be written or defined in term of its previous result to solve a problem using recursion.	4.	It is not necessary to define a problem in terms of its previous result to solve using iteration.
5.	All problems cannot be solved using recursion.	5.	All problems can be solved using iteration.

STORAGE CLASSES

In language C the storage class is mainly used for the scope of the variable within the program. There is four such storage classes are given below: There are four storage classes in 'C'

a. Automatic Storage Class
b. Register Storage Class
c. Static Storage Class
d. External Storage Class

Storage class	**Storage**	**Default initial value**	**Scope**	**Life**
AUTOMATIC (Auto)	Memory	An unpredictable value, which is often called a garbage value.	Local to block in which the variable is defined.	Till the control remains within the block in which the variable is defined.
REGISTER (register)	CPU register	Garbage value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined.
STATIC (Static)	Memory	Zero (0)	Local to the block in which the variable is defined	Till the control remains within the block in
EXTERNAL (Extern)	Memory	Zero(0)	Global	As long as the program execution doesn't come to an end.

Table: Storage Classes

Examples

a) Auto Variable

```
main()
{
        auto int i=1;
 {
        auto int i=2;
        {
        auto int i=3;
        printf("%d ",i);
        }
        printf("%d ",i);
        }
        printf("%d ",i);
}
```

Output: 3 2 1

b) Register variable:

```
main()
{
        register int i;
        for(i=0; i<5;i++)
        printf("%d ",i);
 }
```

Output: 0 1 2 3 4

c) Static variable

```
main()
 {
        inct();
        inct();
        inct();
 }
 void inct()
 {
        static int i=1;
        printf("%d",i);
        i=i+1;
 }
```

Output: 1 2 3

d) External variable:

```
extern int x=10;
main()
{
        int x=20;
        printf("%d\n",n);
        display ( );
}
display()
{
printf("%d",x);
}
```

Output: 20

```
        10
```

ARRAYS

Array is a collection of same data type.

Array is the contiguous allocation of memory having the common name. To identify each element on the array, an index number called subscription is defined. It is numbered from 0 to (n-1) where the array contains n elements. An Array is also known a subscripted variable.

The following are the points to be noted:

1. An array is a collection of similar elements.
2. The first element of the array is referred with subscript (array index) 0 (zero) and the subscript of the last element is 1 less than the size of the array.
3. Before using an array, its type and dimension must be declared.
4. Its elements are always stored in contiguous memory locations.
5. Without initialization, array contains garbage value.
6. At the time of initialization of array, dimension is optional.

General Syntax for Array Declaration:

<data type> <array_name> [size of array-1][size-2], ..., [size-n];

Initialization at the time of declaration:

<data type> < array_name> [ ] = {value –1, value-2, value-3, ..., value-n};

Example

Declaration of 1 dimension (I-D) array

int a[5];

a[0]	a[1]	a[2]	a[3]	a[4]

Example

Declaration of 1 dimension (I-D) array with initialization

int a[5]={2,4,8,67,22}; or int a[ ]={2,4,8,67,22};

2	**4**	**8**	**67**	**22**
a[0]	a[1]	a[2]	a[3]	a[4]

Example

Declaration of 2-D Array

int a[3][5];

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Example

Declaration of 2-D Array with initialization

int a[3][5]={{3,7,9,5,4},{54,9,94,0,100},{85,2,3,5,84}};

3	7	9	5	4
54	9	94	0	100
85	2	3	5	84

ASSIGNMENT

Syntax and example of

- ¤ Declaration of 3D array
  Hints: int a[2][3][4];

PASSING ARRAYS TO FUNCTIONS

Passing Arrays

- ¤ To pass an array argument to a function, specify the name of the array without any brackets

```
float list[100];
.....
avg = average(list, n);
```

- ¤ The array name is written with an empty square bracket in the formal parameter declaration. float average(float x[], int n){}
- ¤ Name of array is address of first element

Passing Array Elements

– Passed by call-by-value

– Pass subscripted name (i.e., list [3]) to function

/* **function prototype** */

```
float average(float x[], int n);
int main()
{
        int n;
        float avg;
        float list[100];
        .....
        avg = average(list,n);
        .....
}
```

/* **function definition** */

```
float average(float x[], int n)
{
        .....
        }
```

Arrays are always passed by reference

- ¤ Arrays are passed by reference
- ¤ Name of array is treated as the address of the first element in the function
    - Hence it actually becomes a pointer to the first element of the array in the function
- ¤ Function knows where the array is stored
    - Can modify original array elements passed

```
void modify(int b, int c[]);
main() {
int b = 2;
int i, c[] = { 10, 20, 30 };
modify(b,c);
printf("b = %d\n", b);
for (i = 0; i < 3; i++)
printf("c[%d] = %d\n", i, c[i]);
}
void modify(int b, int c[])
{
int i;
b = -999;
for (i = 0; i < 3; i++)
c[i] = -9;
}
```

ARRAYS AND STRING

You can create array of strings using a two-dimensional character array

char months[12][10];

¬ Left dimension determines the number of strings, and right dimension specifies the maximum length of each string

¬ Now you can use the array months to store 12 strings each of which can have a maximum of 10 characters (including the null)

¬ To access an individual string, you specify only the left subscript

puts(months[2]); prints the third month

Example

```
char months[12][10] =
{
"January",
"February",
"March",
"April",
"May",
"June",
"July",
"August",
"September",
"October",
"November",
"December“
};
printf("%s\n", months[5]);
```

months[0]	J	a	n	u	a	r	y	\0		
months[1]	F	e	b	r	u	a	r	y	\0	
months[2]	M	a	r	c	h	\0				
months[3]	A	p	r	i	l	\0				
months[4]	M	a	y	\0						
months[5]	J	u	n	e	\0					
months[6]	J	u	l	y	\0					
months[7]	A	u	g	u	s	t	\0			
months[8]	S	e	p	t	e	m	b	e	r	\0
months[9]	O	c	t	o	b	e	r	\0		
months[10]	N	o	v	e	m	b	e	r	\0	
months[11]	D	e	c	e	m	b	e	r	\0	

String Functions

Strncpy

Where n indicates number of characters

Syntax: strncpy (destination, source, n);

- destination – string variable
- source – string variable or string constant
- n - number of characters

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char name[10];
strncpy (name, "Sallu", 6);
printf("First string %s",name);
strncpy(name,"sallu",2);
name[2]='\0'; /* 1 dimensional array[]
[0][1][2][s][a][null char]*/
/* Null character helps to prevent garbage value….*/
/*remove and check null character.*/
printf("Second string %s",name);
getch();
}
```

Strlen ()

Syntax

strlen (string variable);

String length that strlen function return should be an integer value.

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

main()

{

        int l;

        char name[10];

        printf("Enter a string");

        scanf("%s", name);

        l=strlen(name);

        printf("length is %d",l);

        getch();

}
```

DYNAMIC MEMORY ALLOCATION

The exact size of array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into an executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not have any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

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()

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.

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

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 n,i,*ptr,sum=0;

    printf("Enter number of elements: ");

    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc

    if(ptr==NULL)

    {

        printf("Error! memory not allocated.");

        exit(0);

    }

    printf("Enter elements of array: ");

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

    {

        scanf("%d",ptr+i);

        sum+=*(ptr+i);

    }

    printf("Sum=%d",sum);

    free(ptr);

    return 0;

}
```

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

```
#include <stdio.h>

#include <stdlib.h>

int main(){

    int n,i,*ptr,sum=0;

    printf("Enter number of elements: ");

    scanf("%d",&n);

    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient, then, you can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, *ptr* is reallocated with size of newsize.

Illustrative Program

```
#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;

}
```

STRUCTURES

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
- Structures are called “records” in some languages, notably Pascal.
- Structures help to organize complicated data, particularly in large programs.
- They permit a group of related variables to be treated as a unit instead of as separate entities.
- A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together
- Structures are one of the ways to create a custom data type in C.
- Variables that make up the structure are called ***members,*** also commonly referred to as ***elements*** or ***fields***
- A structure declaration is identified by the struct keyword, followed by an identifier known as ***structure tag.***

Example of structure declaration

A structure declaration to represent information about a student’s name and other details

```
struct student
{
int id;
char name[30];
char sex;
float marks[7];
float total;
float per;
int result;
};
```

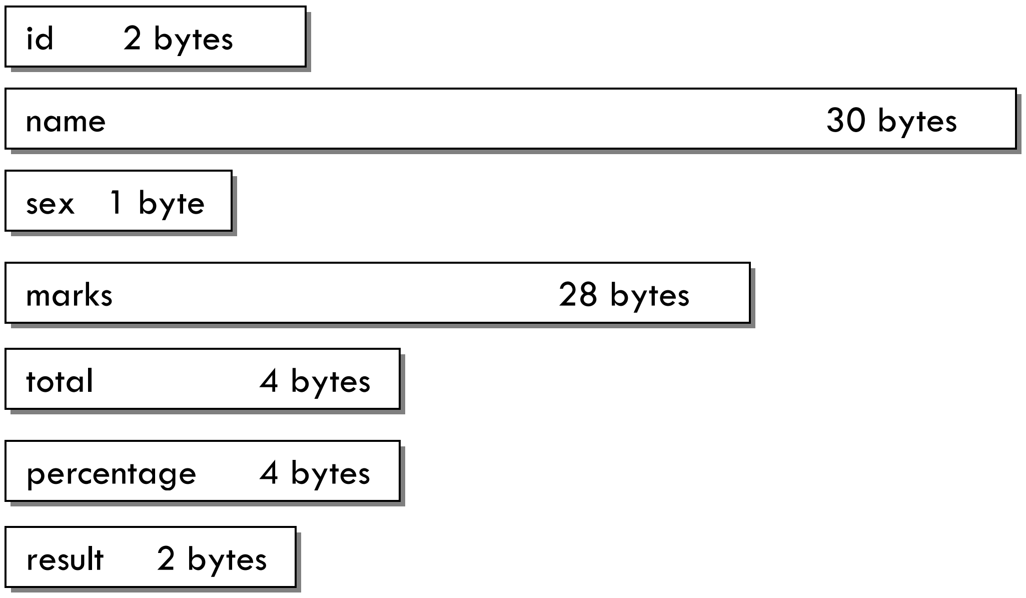
Here, struct is the required keyword and student is the structure tag used to identify this structure

Structure Variables

- A structure declaration only defines the form of the data; it does not allocate memory
- A *structure variable* is a variable of a structure type
- To declare a variable of type student, defined earlier, write **struct student st;**
- This declares a variable of type struct student called st. Thus, student describes the form of a structure (its type), and st is an instance (a variable) of the structure

Structure Variable in Memory

When a structure variable (such as st) is declared, the compiler automatically allocates sufficient memory to accommodate all of its members



The structure variable st in memory, although not shown, all the members are stored contiguously in memory.

You **can also declare one or more variable** when you declare a structure. For example:

```
struct student
{
int id;
char name[30];
char sex;
float marks[7];
float total;
float per;
int result;
} a, b, c;
```

– defines a structure type called student and declares three variables a, b and c of that type

• Each structure variable (a, b, and c) contains its own copies of the structure's members

If you only need one structure variable, **the structure tag can be omitted.** For example:

```
struct
{
int id;
char name[30];
char sex;
float marks[7];
float total;
float per;
int result;
} st;
```

– declares one variable named st as defined by the structure preceding it

General Form of Structure Declaration

The general form of a structure declaration is

```
struct tag
{
type member-name;
type member-name;
type member-name;
...
} structure-variables;
```

Where either tag or structure-variables may be omitted, but not both

- ***A structure member or tag and an ordinary variable can have the same name without conflict***
- The same member names may occur in different structures.

Accessing Structure Members

A structure member can be accessed by writing

variable.member where variable refers to the name of structure-type variable, and member refers to the name of a member within a structure

The ***period*** (.) that separates the structure variable name from the member name is called the ***dot*** or ***structure – member operator.***

- For accessing name stu.name we can also use scanf.
- We have to follow the sequence of the member data

Example:

```
Struct student
{
int roll;
char name[10];
}stu = {101, "RAMAN"};
printf("Student roll no is %d", stu.roll);
stu.roll=102;  ⟶ for accessing single member.
```

Example of Structure

```
#include<stdio.h>
#include<conio.h>
struct student
{
        int roll;  ───────┐
                          ├── members  } structure definition
        char name;  ──────┘
} stu1 = {101, "RAJAN"};
  ↓           ↘
Variable        Should be sequential
main()
{
        struct student stu2;
        printf("Roll no is %d",stu1.roll);
        printf("student name is %s",stu1.name);
        printf("Enter roll no for second student");
        scanf("%d",&stu2.roll);
        printf("Enter name for second student");
        scanf("%d",&stu2.name);
        printf("student Roll no: %d ",stu2.roll);
        printf("student name: %s ",stu2.name);
getch();
}
```

Nested Structures

Since structure is a custom type, you can define structure variable as a member of another structure.

```
struct time {
int hrs, mins;
};
struct date {
int m,d,y;
};
struct flightschedule {
int flightno;
struct time departuretime;
struct time arrivaltime;
struct date scheduledate;
};
```

Here the structure variables departuretime, arrivaltime, and scheduletime are members of the structure flightschedule. These are said to be nested within the structure flightschedule. The declaration of time and date must precede the declaration of flightschedule.

NESTED STRUCTURE IN C [STRUCTURE WITHIN STRUCTURE]

Syntax:

```
struct   structure_tag   ————→ outer
{
        <data_type> member1;
        -----------------
        struct structure_tag   ————→ inner
        {
                <data_type> member1;
                ---------------
        } inner_var;
} outer_var;
```

How to access inner and outer variable?

outer_variable . member_name ;

outer_variable . inner_variable . member_name ;

Because inner variable is defined inside outer structure

/***Program for nested structure in C***/
/* Two structures: **outer**→ students name and roll **inner**→ students subject code and marks.*/

```
#include<stdio.h>
#include<conio.h>
struct student
{
        int roll;
        char name;
        struct stu_mark;
                {
                        char sub[10];
                        int m;
                } mark; ————→ for accessing inner variable
 } ; ———→ outer variable will be created inside main function
main()
{
        struct student stu;
        printf("Enter roll no:");
        scanf("%d",&stu . roll);
        printf("Enter student name:");
        scanf("%s",&stu . name);
        printf("Enter subject code:");
        scanf("%s",&stu . mark . sub);
        printf("Enter mark:");
        scanf("%d",&stu . mark . m);
        printf("Student details:");
        printf("Roll no %d", stu . roll);
        printf("Student name %s", stu . name);
        printf("Subject code  %s", stu . mark . sub);
        printf("Mark  %d", stu . mark . m);
getch();
}
```

Array of Structures

- Since you can create an array of any valid type, it is possible to define an array of structures; i.e., an array in which each element is a structure.
- To declare an array of structures, you must declare a structure and then declare an array variable of that type.

```
Array:    int marks[10];
Example:
Struct student
{
        ---------Roll_no;
        ---------name;
} stu[5]; /* array of structures */
```

To access a specific structure, you index the array name, stu.

For example to print the Roll_no of 3rd student, write:

printf("%d", stu[2].Roll_no);

Illustrative Program for Array of Structures

```
#include<stdio.h>

#include<conio.h>

struct student
{
        int roll;
        char name[10];
};
main()

{                                               array of structure

        struct student stu[5];

        int i;

        printf("Enter Details for five students");

        for(i=0; i<5; i++)
                {                                                        0+1
                printf("Enter roll no for %d student",i+1);

                scanf("%d",&stu[i].roll);

                printf("Enter name for %d student",i+1);

                scanf("%s",&stu[i].name);

                                                 Variable subscript ∵ it is array

                }
        printf("Details for students:");

        for(i=0; i<5; i++)
                {
                printf("Detail for %d student",i+1);

                printf("Roll no %d",stu[i].roll);

                printf("Name %s",stu[i].name);


                }
getch();        }
```

PASSING STRUCTURES TO FUNCTIONS |Passing Structure Members to Functions

¬ Individual structure members can be passed to functions

¬ When a structure member is passed to the function, the value of the member is passed

¬ In the function, each structure member is treated the same as an ordinary single - valued variable

```
float adjust(char name[], int acct_no, float
balance, char type)
{
        ...
}
main()
{
        struct bankaccount acc;
        ...
        /* pass individual members to the
        function */
acc.balance = adjust(acc.name,acc.acct_no,
acc.balance,acc.balance);
        ...
}
```

Note for acc.name**, it is the address of** acc.name[0] **that is passed. For other members, it is the value of the members that are passed to function. It is irrelevant in the function** adjust **that structure members were passed**

SELF-REFERENTIAL STRUCTURES

¤ A self-referential structure is a structure in which at least one of the member is a pointer to the parent structure type

¤ ***For example***

```
struct node {
```

```
char name[40];
struct node *next;
};
```

¤ This structure contains two members: a 40-element character array, called item and a pointer to a structure of the same type (a pointer to a structure of type node), called next

¤ Useful in applications that involve dynamic data structures, such as lists and trees

UNIONS

¤ Memory that contains a variety of objects over time.

¤ Only contains one data member at a time.

¤ Members of a union share space.

¤ Conserves storage.

¤ Only the last data member defined can be accessed.

¤ The size required by a union variable is the size required by the member requiring the largest size.

Union Declarations

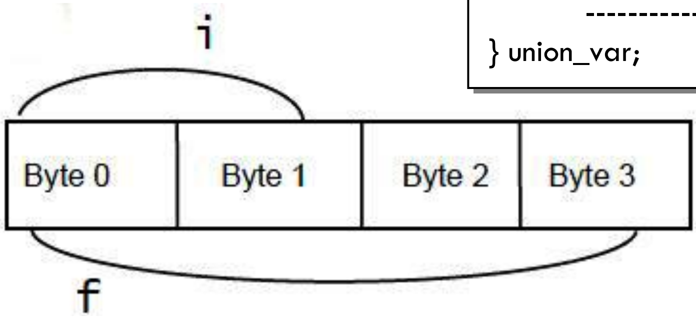
Syntax:
Similar to structure, difference is in keyword struct and union.

```
union tag_name
{
        <data_type>
member1;
        <data_type>
member2;
        --------------------
        --------------------
} union_var;
```

Same as struct

```
        union number {
        int i;
        float f;
        };
union number value;
```

• The variable value in memory



Illustrative Program for UNION

```
#include<stdio.h>
#include<conio.h>
union student
{
        int roll;
        char name[10];
} ;
main()
{
        union student stu;
        printf("Enter roll no.");
        scanf("%d", &stu . roll);
         printf("Enter name.");
        scanf("%s", &stu . name);
        printf("Student details");
        printf("Roll no %d",stu . roll);
        printf("student name %s",stu . name)
getch();
}
```

Output | Explanation of Program

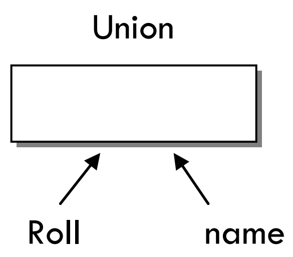
```
Roll no 101
Student name raj
```

Output

Roll no roll no may be garbage value (unexpected value)

Student name raj

Reason is that: union members are sharing one memory location.



Roll no: 2 byte space

Name: 10 byte

First roll is stored then name is stored. Roll no is overlapped by roll.

/* WAP to make clear the size of union and structure.*/

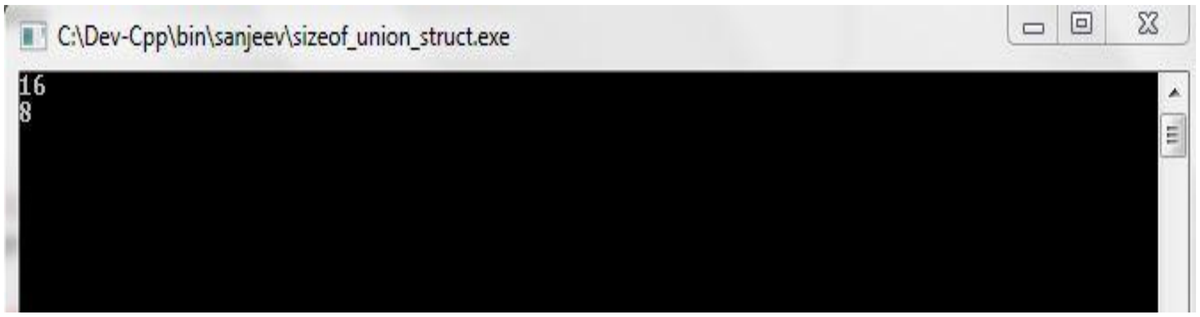
```
#include<stdio.h>
#include<conio.h>
```

```
int main(){
   struct Person{
         int age;
         char name[6];
         float height;
         };

   union Student{
        int age;
        char name[6];
         float height;
         };
        printf("%d\n", sizeof(struct Person));
        printf("%d\n", sizeof(union Student));
   getch();
   return 0;  }
```

Difference between structure and union

- Similar to structure, difference is in keyword: struct and union.
- Member of union share single memory location.
- In structure each member shares different memory location.
- Structure size → summation of members size, member has different memory location, one member value is not affected and can affect any other.
- Union share common memory location.



C:\Dev-Cpp\bin\sanjeev\sizeof_union_struct.exe

```
16
8
```

POINTER

The knowledge of memory address is necessary before understanding pointer. All computer have primary memory, also known as **RAM** (Random access Memory). **RAM** holds the programs that the computer is currently running along with the data (i.e. the memory when the program is executed). The computer memory (RAM) is divided into a number of small location or units. Each location is represented by some unique number (0 to 255), which is known as a byte. A *char* data is one byte in size and hence needs two memory location of the memory. Similarly, *integer* data is two byte in size and hence needs two memory location of memory. The smallest unit for memory address is bit (**bi**nary digi**t**) i.e. 0 or 1.

Each byte in memory is associated with a unique address. An address is an integer labeling a byte in the memory. An integer is always a positive value that range from zero to some positive integer constants corresponding to the last location in the memory. Thus a computer having 1 GB RAM has 1024 * 1024 * 1024 i.e. 1073741824 bytes and these 1073741824 bytes are represented by 1073741824 different addresses. For Example, the memory address 65524 represents a byte in memory and it can store data of one byte.

Info tip

The key concepts:
8 bits = 1 bytes. For Ex: 1001 1111 is one byte
1024 bytes = 1 kilo bytes (KB)
1024 KB = 1 Mega Bytes (MB)
1025 MB = 1 Giga Bytes (GB)

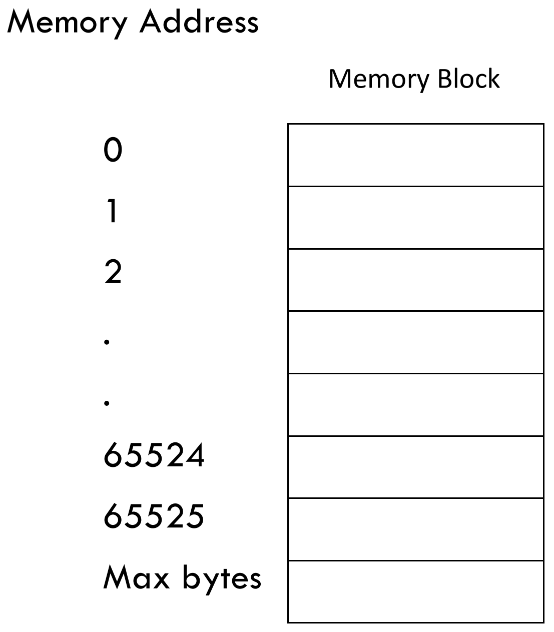
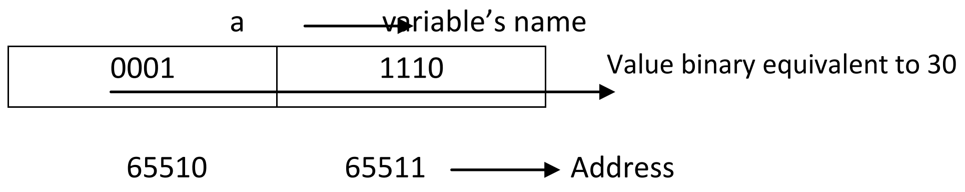


Fig: Memory Addresses in Memory

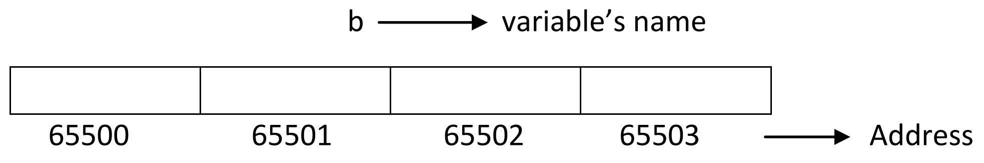
Every variable in a C program is assigned a space in memory. When a variable is declared, it tells computer the type of variable and name of the variable. According to type of variable declared, the required memory locations are received. For Example, *int* requires two bytes, *float* requires four bytes and *char* requires one bytes.

For Ex:

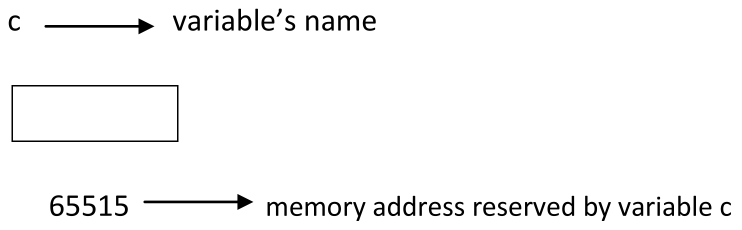
int a = 30;



float b;



char c;

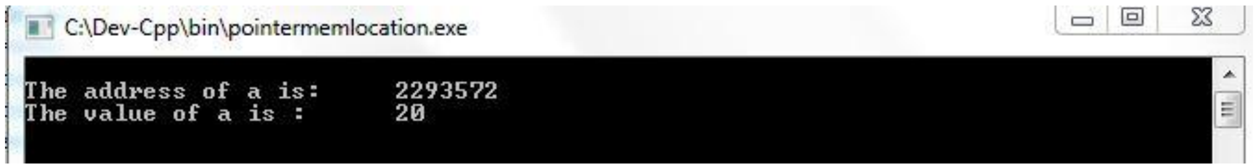


/* WAP to display memory location reserved by a variable.*/

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a;
    a=20;
    printf("\nThe address of a is:\t %u",&a);
    printf("\nThe value of a is :\t %d ",a);
    getch();  }
```

Explanation of the program

Here, &a denote address of variable a. The variable a takes two bytes in memory as it is of type int. It takes two memory locations 65524 and 65525. The control string u (unsigned) is used to display address of the variable, as the value of address is positive integer.


C:\Dev-Cpp\bin\pointermemlocation.exe

```
The address of a is:     2293572
The value of a is :      20
```

What is a pointer? Why we use pointers in C?

Pointer is a special type of variable which is used to hold the address of variable. If we use the pointer to operate the variable, then it is quite faster than the ordinary operation. Moreover uses of pointer minimize the complexity of the program. There are two types of symbols which are used to handle the pointer, & and *. Like a postcard, a pointer holds an address of a memory location.

Pointer is an address variable that contains/holds the address of another variable. The address is location of another variable in the memory. There are many reasons for using then pointer, some of which are as follows:

- A pointer allows passing variables, arrays, functions, strings and structures as functions arguments.
- It supports dynamic memory allocation and de-allocation of memory segments.
- With the help of pointers variables can be swapped without physically moving them.
- Since pointer is directly related to the address of the variable, so the efficiency or speed of the execution of the program will be increased.
- The use of pointer reduces the length of the program.

Some points must be considered while using pointers

a) One pointer can hold only one address at a time.

b) Before use, pointer must be defined.

c) At the time of definition, the symbol * must be used as prefix.

d) Pointer holds only address, not the value.

e) Pointer when used in program after definition, then

i) * means „value of"

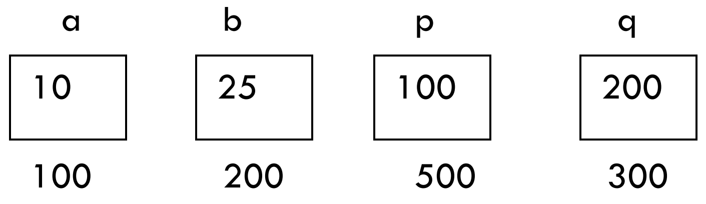
ii) & means „address of"

f) Pointer arithmetic is possible for addition and subtraction, but it has no meaning of multiplication and division.

The following example illustrates the above aspects.

int a,b,*p,*q;

a=10; b=25; p=&a; q=&b; [address of a is assigned to p, address of b assigned to q]



Expression	Output
a	10
b	25
&a	100
&b	200
p	100
q	200
&p	500
&q	300
&(&a)	500
&(&b)	300
*(&a)	10
*(&b)	25

Example: Program to add of two numbers using pointer.

```
main()
{
        int a,b,*p,*q,c;
         p=&a;
         q=&b;
         printf ("Enter two numbers : ");
         scanf("%d %d",p,q);
         c=*p+*q;
        printf ("The sum is = %d",c);       }
```

Pointer Arithmetic:

As a pointer holds the memory address of variable, some arithmetic operations can be performed with pointers. C and C++ support four arithmetic operations that can be used with pointers, such as:

Purpose	**Symbol**
Addition	+
Subtraction	-
Incrimination	++
Dicriminattion	-

Pointers are variables. They are not integers, but they can be as unsigned integers. The conversion specifier for a pointer is added and subtracted. **For Ex:**

Ptr ++	cause the pointer to be incremented, but not its value by 1.
Ptr --	causes the pointer to be decremented, but not its value by 1.

According to the data type declared to that pointer variable if arithmetic operation is done then values or content will be incremented or decremented as per data type is chosen.

/* **WAP to increment the pointer's address.***/

```
#include<stdio.h>
#include<conio.h>
main()
{
        int value, *ptr;
        value=120;
        ptr=&value;
        printf("\naddress of ptr is:%d",ptr);
        ptr++;
        printf("\naddress of ptr is %d",ptr);
        getch();
}
```

ARRAY OF POINTERS

An array of pointers can be defined as

*data_type *pointer_name[size];*

for ex:

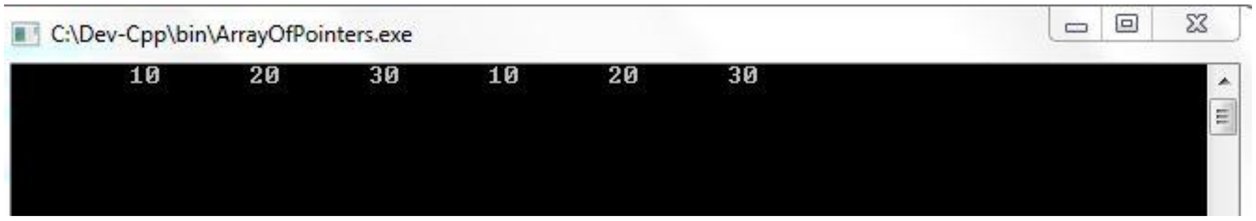
```
int  *p[10];
```

This declares an array of 10 pointers, each of which points to an integer. The first pointer is called p[0], the second is called p[1] and so on up to p[9]. Initially these pointers are uninitialized and they can be used as below:

```
int a=10, b=100, c=1000;
p[0]=&a;
p[1]=&b;
p[2]=&c; and so on.
```

/* WAP to illustrate array of pointers.*/

```
#include<stdio.h>
#include<conio.h>
main()
{
        /* for normal variables*/
        int marks[]={10,20,30};
        /*Array of pointers.*/
        int  *point[3],i;
        for(i=0 ;  i<3 ; i++)
        {
                printf("\t%d",marks[i]);
                point[i]=&marks[i];
    }
        /* now printing the values by using pointer.*/
        for(i=0; i<3; i++)
        {
                printf("\t%d",*point[i]);
        }
getch();
}
```



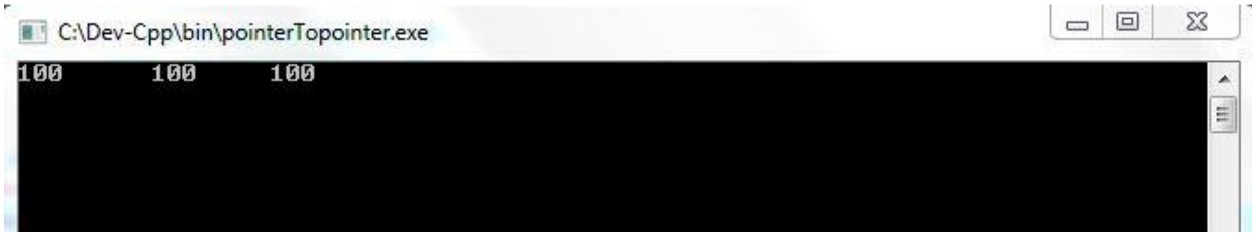
C:\Dev-Cpp\bin\ArrayOfPointers.exe

```
        10        20        30        10        20        30
```

POINTER TO POINTER IN C

/* WAP to illustrate the concept of normal pointer and pointer to pointer in C program.*/

```
#include<stdio.h>
#include<conio.h>
main()
{
        int no = 100;
        int *p;                 //  *p is pointing to no
        int **p1;               //   **p1 is pointing to *p.
        p = &no;
        p1 = &p;
        printf("%d",no);           // output will be 100
        printf("%d", *p);         // output will be 100
        printf("%d", **p1);   // output will be 100
getch();        }
```



C:\Dev-Cpp\bin\pointerTopointer.exe

```
100       100       100
```

POINTERS TO FUNCTIONS IN C

Syntax:

<function_return_type> (<*pointer_name>) (function_argument_type)

Example:

void (*ptr) (int,int);

Syntax:

Pointer_name = &function_name;   Ex: ptr = &add;

Pointer_name = function_name;   Ex: ptr = add;

Calling functions through pointer

Example:

Ptr (10 , 20);

(*ptr)(10 , 20);

/* WAP to illustrate pointers to functions in C.*/

```
#include<stdio.h>

#include<conio.h>

void add (int x, int y)

{

        printf("first value: %d",x);

        printf("second value: %d",y);

        printf("Addition is: %d",x+y);

}

main()

{

        int a , b;
        void (*ptr) (int , int);

        ptr = add;

        printf ("Enter first no:");

        scanf ("%d",&a);

        printf ("Enter second no:");

        scanf ("%d",&b);

ptr (a , b); // function is being called by using pointer

getch();

}
```

STRUCTURES AND POINTERS

Pointer to a Structure Variable

- ¤ C allows pointers to structures just as it allows pointers to any other type of variable.
- ¤ Like other pointers, structure pointers are declared by placing * in front of the structure variable name.
- ¤ *For example*

  struct banckaccount *pAccount; declares pAccount as pointer to a structure variable of type struct bankaccount.

Using Structure Pointers

- ¤ To initialize a structure pointer, use the & operator to get the address of a structure variable.

  ```
  struct bankaccount acc, *pAcc;
  pAcc = &acc;
  ```
- ¤ Now pAcc is a pointer to a structure of type bankaccount, and *pAcc is pointed structure variable (acc).

¤ (*pAcc).acct_no is the acct_no member of the pointed structure variable, acc

- The parentheses are necessary in (*pAcc).acct_no because the precedence of the structure member operator . is higher then *
- The expression *pAcc.acct_no means *(pAcc.acct_no), which is illegal here because acct_no is not a pointer.

The arrow pointer

Pointers to structures are so frequently used that an alternative notation is provided as a shorthand

• If p is a pointer to a structure, then

– p->member-of-structure

– refers to the particular member

• The ->, usually called the arrow operator, is used to access a structure member through a pointer to structure

• Hence, pAcc->acct_no is same as (*pAcc).acct_no

Using Structure Pointer

```
struct bankaccount {
int acct_no;
char acct_type;
char name[80];
float balance;    };
main()
{
struct bankaccount acc = {1001, 'C', "Binod
Chapagain",12000.0};
struct bankaccount *pAcc;
pAcc = &acc;
printf("Account No: %d\n", pAcc->acct_no);
printf("Account Type: '%c'\n", pAcc->acct_type);
printf("Account Holder'name: %s\n", pAcc->name);
printf("Current Balance: %f\n", pAcc->balance);
}
```

Use of Structure Pointers

There are two primary **uses** for structure pointers:

- To pass a structure to a function using a call by reference
- To create linked lists and other dynamic data structures that relies on dynamic allocation.

There is one major **drawback** to passing structure variables to functions: the overhead needed to creating a copy and passing it to the function.

DATA FILES | FILES

In C programming, file is a place on disk where groups of related data are stored.

Why files are needed?

When the program is terminated, the entire data is lost in C programming. If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created, these information can be accessed using few commands.
There are large numbers of functions to handle file I/O in C language. In this tutorial, you will learn to handle standard I/O (High level file I/O functions) in C.
High level file I/O functions can be categorized as:

1. Text file
2. Binary file

Text Stream	Binary Stream
¤ A **text stream** consists of *sequence of characters*, such as text data being sent to the screen.	¤ A **binary stream** is a *sequence of bytes*
¤ Text streams are organized into lines, which can be up to 255 characters long and are terminated by an end-of-line, or newline, character.	¤ A binary stream can handle any sort of data, including, but not limited to, text data
¤ Certain characters in a text stream are recognized as having special meaning, such as the newline character.	¤ Bytes of data in a binary stream aren't translated or interpreted in any special way; they are read and written exactly as-is.
	¤ In text stream, end of file is determined by a special character having ASCII value 26, but in binary stream end of file is determined by the directory listing of host environment

File Operations

1. Creating a new file
2. Opening an existing file
3. Reading from and writing information to a file
4. Closing a file

Commonly Used C File-System Functions

Name	Functions
fopen	Opens a file
fclose	Closes a file
putc, fputc	Writes a character to a file
getc, fgetc	Reads a character from file
fgets	Reads a string from a file
fputs	Writes a string to a file
fprintf	Is to a file what printf is to the console
fscanf	Is to a file what scanf is to the console
feof	Returns true if end-of-file is reached
ferror	Returns true if an error has occurred
fflush	Flushes a file

Working with file

While working with file, you need to declare a pointer of type file. This declaration is needed for communication between file and program.

```
FILE *ptr;
```

Opening a file

Opening a file is performed using library function fopen(). The syntax for opening a file in standard I/O is:

```
ptr=fopen("fileopen","mode")

For Example:
fopen("C:\\cprogram\program.txt","w");

/* -------------------------------------------------------- */
 C:\\cprogram\program.txt is the location to create file.
 "w" represents the mode for writing.
                    /* -------------------------------------------------------- */
```

Here, the program.txt file is opened for writing mode.

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e. Data is added to end of file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.

Table: Opening Modes in Standard I/O

Closing a File

The file should be closed after reading/writing of a file. Closing a file is performed using library function fclose().

fclose(ptr); //ptr is the file pointer associated with file to be closed.

```
#include <stdio.h>

int main()

{

  int n;

  FILE *fptr;

  fptr=fopen("C:\\program.txt","w");

  if(fptr==NULL){

     printf("Error!");

     exit(1);
  }
  printf("Enter n: ");
  scanf("%d",&n);
  fprintf(fptr,"%d",n);
  fclose(fptr);
  return 0;
}
```

Explanation of the program

This program takes the number from user and stores in file. After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open that file, you can see the integer you entered.

Similarly, fscanf() can be used to read data from file.

Reading from file

```
#include <stdio.h>
#include<conio.h>
int main()
{
  int n;
  FILE *fptr;
  if ((fptr=fopen("c:\\program.txt","r"))==NULL){
     printf("Error! opening file");
     exit(1);        /* Program exits if file pointer returns NULL. */
  }
  fscanf(fptr,"%d",&n);
  printf("Value of n=%d",n);
  fclose(fptr);
  getch();
  return 0;
}
```

Explanation of the program

If you have run program above to write in file successfully, you can get the integer back entered in that program using this program.

Other functions like fgetchar(), fputc() etc. can be used in similar way.

ASSIGNMENT | EXERCISES | SOME IMPORTANT QUESTIONS FROM EXAM POINT OF VIEW

INTRODUCTION TO C PROGRAMMING

1. Write a program in C that displays “Let us make C easy”.
2. WAP to multiply two numbers and display product. The program should ask two numbers from keyboard.
3. WAP that will convert temperature in Centigrade into Fahrenheit and vice - versa.
4. Write C programs that ask the name of student and marks obtained by him/her in 5 subjects. Display the percentage of the students assuming full marks 100 for each subject.
5. WAP to illustrate the use of escape sequence.

CONTROL STRUCTURE

1. Write a program that asks user for a number and test the number whether it is positive or negative. Display appropriate message to the user.
2. WAP to test a number for even or odd. Display appropriate message to the user.
3. WAP to find the largest of three numbers:
   a. Using nested *if…else*
   b. Without use of nested *if…else*
4. WAP to display series of numbers or symbols as given below:

```
*
*    *
*    *    *
*    *    *    *
*    *    *    *    *
```

5. WAP that reads two numbers and an arithmetic operator (one of +, -, *, /, %). Perform respective operation using switch statement.

FUNCTION

1. Write a function to convert a lowercase character into uppercase and vie – versa using call by reference.
2. Write a recursive function to generate Fibonacci series.

3. WAP to calculate factorial of a number using recursive method.
4. Define a function named factorial () to calculate factorial of n number and then write a program that uses this function factorial () to calculate combination and permutation.
5. WAP to read 3 digit number and check whether the number is Armstrong or not.

ARRAY AND POINTER

1. WAP to read a string and check for palindrome.
2. WAP that sort the alphabet entered by the user in alphabetical order.
3. WAP to add two m * n matrices using pointer.
4. WAP to pass 7 days temperature and calculate average temperature, maximum temperature and minimum temperature using array.
5. WAP to input name, roll no , and total marks obtained by the student for your class then generate the merit list.
6. WAP to access the array element using pointer.

STRUCTURE AND DATA FILES

1. WAP to display the name and age of n students using the concept of both structure and pointer.
2. Create a structure named *data* that has *day*, *month* and *year* as its members. Include the structure as a member in another structure named *Employee* which has *name, id* and *salary* as other members. Use this structure to read and display employee's name, id, date of birth and salary.
3. Create a file named "test.txt" and write some text "Welcome to Welhams College" to the file.
4. WAP to open a file named inventory.txt and store in it the following data

Product Name	**Qty**	**Rate**
AAA	3	50
BBB	4	70
CCC	6	12

MINOR PROJECT

WAP that allows a user to *enter name, age sex, number of songs* **recorded for duet singers using a file. The program should also have the provision for** *editing, searching, deleting and listing* **the details of singer. The program should be menu driven.**

Books for study and Reference

1. Kanetkar Y., Let us C, BPB Pub., New Delhi, 1999.
2. H. Schildt, C: The Complete Reference, 4th Edition, TMH Edition, 2000
3. B.W. Kernighan and D.M.Ritchie, The C Programming Language, 2nd Edition, PHI, 1988.

Note | Suggestions

If any suggestions……… mail to – gtssanjeev@gmail.com. Thanks for your co – operation……………..