PRATHYUSHA

ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REGULATION R2021

I YEAR - II SEMESTER

CS3251 – PROGRAMMING IN C

CS3251 **PROGRAMMING IN C**

COURSE OBJECTIVES:

- To understand the constructs of C Language.
- To develop C Programs using basic programming constructs
- To develop C programs using arrays and strings
- To develop modular applications in C using functions
- To develop applications in C using pointers and structures
- To do input/output and file handling in C

UNIT I **BASICS OF C PROGRAMMING**

Introduction to programming paradigms – Applications of C Language - Structure of C program - C programming: Data Types - Constants – Enumeration Constants - Keywords – Operators: Precedence and Associativity - Expressions - Input/Output statements, Assignment statements – Decision making statements - Switch statement - Looping statements – Preprocessor directives - Compilation process

UNIT II **ARRAYS AND STRINGS**

Introduction to Arrays: Declaration, Initialization – One dimensional array – Two dimensional arrays - String operations: length, compare, concatenate, copy – Selection sort, linear and binary search.

UNIT III **FUNCTIONS AND POINTERS**

Modular programming - Function prototype, function definition, function call, Built-in functions (string functions, math functions) – Recursion, Binary Search using recursive functions – Pointers – Pointer operators – Pointer arithmetic – Arrays and pointers – Array of pointers – Parameter passing: Pass by value, Pass by reference.

UNIT IV **STRUCTURES AND UNION**

Structure - Nested structures – Pointer and Structures – Array of structures – Self referential structures – Dynamic memory allocation - Singly linked list – typedef – Union - Storage classes and Visibility.

UNIT V **FILE PROCESSING**

Files – Types of file processing: Sequential access, Random access – Sequential access file - Random access file - Command line arguments.

COURSE OUTCOMES: Upon completion of the course, the students will be able to

CO1: Demonstrate knowledge on C Programming constructs

CO2: Develop simple applications in C using basic constructs

CO3: Design and implement applications using arrays and strings

CO4: Develop and implement modular applications in C using functions.

CO5: Develop applications in C using structures and pointers.

CO6: Design applications using sequential and random access file processing.

TOTAL : 45 PERIODS

TEXT BOOKS:

1. ReemaThareja, “Programming in C”, Oxford University Press, Second Edition, 2016.

2. Kernighan, B.W and Ritchie,D.M, “The C Programming language”, Second Edition, Pearson Education, 2015.

REFERENCES:

1. Paul Deitel and Harvey Deitel, “C How to Program with an Introduction to C++”, Eighth edition, Pearson Education, 2018.

2. Yashwant Kanetkar, Let us C, 17th Edition, BPB Publications, 2020.

3. Byron S. Gottfried, “Schaum’s Outline of Theory and Problems of Programming with C”, McGraw-Hill Education, 1996.

4. Pradip Dey, Manas Ghosh, “Computer Fundamentals and Programming in C”, Second Edition, Oxford University Press, 2013.

5. Anita Goel and Ajay Mittal, “Computer Fundamentals and Programming in C”, 1st Edition, Pearson Education, 2013.

UNIT 1

BASICS OF C PROGRAMMING

Introduction to programming paradigms – Applications of C Language - Structure of C program - C programming: Data Types - Constants – Enumeration Constants - Keywords – Operators: Precedence and Associativity - Expressions - Input/Output statements, Assignment statements – Decision making statements - Switch statement - Looping statements – Preprocessor directives - Compilation process .

1. INTRODUCTION TO PROGRAMMING PARADIGMS:

C INTRODUCTION:

The programming language "C" was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. Although C was initially developed for writing system software, today it has become such a popular language that a variety of software programs are written using this language.

The greatest advantage of using C for programming is that it can be easily used on different types of computers. Many other programming languages such as C++ and Java are also based on C which means that you will be able to learn them easily in the future. Today, C is widely used with the UNIX operating system.

PROGRAMMING PARADIGMS:

In computing, a program is a specific set of ordered operation for a computer to perform.The process of developing and implementing various sets of instruction to enable a computer to perform a certain task is called PROGRAMMING.

PROGRAMMING PARADIGMS INCLUDE:

1. **IMPERATIVE PROGRAMMING PARADIGMS:**
   Command show how the computation takes place, step by step. Each step affects the global state of the computation.
2. **STRUCTURED PROGRAMMING PARADIGMS:**
   It is a kind of imperative programming where the control flow is defined by nested loops, conditionals, and subroutines, rather than via gotos. Variables generally local to blocks.
3. **OBJECT ORIENTED PROGRAMMING(OOP) PARADIGMS:**
   It is a programming paradigms based on the concepts of objects, which may contain data, in the form of fields, often known as attributes, and code, in the form of procedures, often known as methods.
4. **DECLARATIVE PROGRAMMING PARADIGMS:**
   The programmer states only what the results should look like , not how to obtain it. No loops, no assignments, etc. Whatever engines that interprets this code is just supposed go gets the desired information and can use whatever approach its wants.
5. **FUNCTIONAL PROGRAMMING PARADIGMS:**
   In functional programming, control flow is expressed by combining functional calls, rather than by assigning values to variables.
6. **PROCEDURAL PROGRAMMING PARADIGMS:**
   This paradigms includes imperative programming with procedure calls.

7. **EVENT DRIVEN PROGRAMMING PARADIGMS**:
   In which the flow of the program is determined by events such as user action(mouse clicks, key presses), sensor output, or message from other program/threads. It is the dominant paradigms used in GUI and other applications that are centred on performing certain action in response to user input.
8. **FLOW DRIVEN PROGRAMMING PARADIGMS:**
   Programming processes communicating with each other over predefined channels.
9. **LOGIC PROGRAMMING PARADIGMS:**
   Here programming is done by specifying a set of facts and rules. An engine infers the answer to question.
10. **CONSTRAINTS PROGRAMMING PARADIGMS:**
    An engine finds the value that meet the constraints.

    One of the characteristics of a language is its support for particular programming paradigms. For example: small talks has direct support for programming in the object oriented way, so it might called an object oriented language.

    Very few language implement a paradigms 100%, when they do, they are "PURE". It is incredibly rare to have a "pure OOP language" or a "pure functional language".

    A lot of language will facilitate programming in one or more paradigms. If a language is purposely designed to allow programming in many paradigms is called a **"multi paradigms language".**

APPLICATION OF C:

1. **OPERATING SYSTEM**
2. **EMBEDDED SYSTEM**
3. **GUI(GRAPHICAL USER INTERFACE)**
4. **NEW PROGRAMMING PLATFORMS**
5. **GOOGLE**
6. **MOZILLA FIREBOX AND THUNDERBIRD**
7. **MYSQL**
8. **COMPILER DESIGN**
9. **ASSEMBLERS**
10. **TEXT EDITORS**
11. **DRIVERS**
12. **NETWORK DEVICES**
13. **GAMING AND ANIMATION**

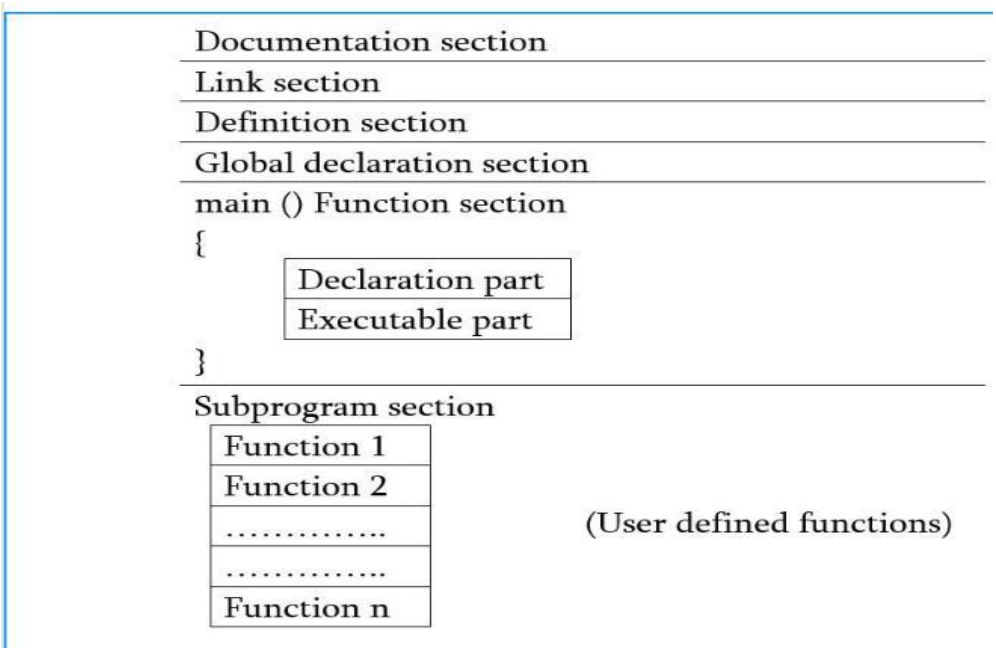
FEATURES OF C PROGRAMMING/ADVANTAGES:

- C is a robust language with rich set of built in function.
- Programs written in c are efficient and fast.
- C is highly portable, programs once written in c can be run on another machine with minor or no modification.
- C is basically a collection of c library functions, we can also create our own function and add it to the c library.
- C is easily extensible.

DISADVANTAGE OF C:

- C doesnot provide OOP.
- There is no concepts of namespace in c.
- C doesnot provides binding or wrapping up of a single unit.
- C doesnot provide constructor and destructor.

STRUCTURE OF C:



Documentation section:

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.

Link section: The link section provides instructions to the compiler to link functions from the system library such as using the #include directive.

Definition section: The definition section defines all symbolic constants such using the #define directive.

Global declaration section: There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions.

Main () function section: Every C program must have one main function section. This section contains two parts; declaration part and executable part.

Declaration part:

The declaration part declares all the variables used in the executable part.

Executable part:

There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

Subprogram section:

If the program is a multi-function program then the subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

All section, except the main () function section may be absent when they are not required.

C PROGRAMMING: DATA-TYPES

A data-type in C programming is a set of values and is determined to act on those values. C provides various types of data-types which allow the programmer to select the appropriate typefor the variable to set its value.

The data-type in a programming language is the collection of data with values having fixed meaning as well as characteristics. Some of them are integer, floating point, character etc. Usually, programming languages specify the range values for given data-type.

C Data Types are used to:

- Identify the type of a variable when it declared.
- Identify the type of the return value of a function.
- Identify the type of a parameter expected by a function.

ANSI C provides three types of data types:

1. Primary(Built-in) Data Types:void, int, char, double and float.
2. Derived Data Types:Array, References, and Pointers.
3. User Defined Data Types:Structure, Union, and Enumeration.

Primary Data Types:

Every C compiler supports five primary data types:

void -As the name suggests it holds no value and is generally used for specifyingthe type of function or what it returns. If the function has a void type, it means that the function will not return any value.
int-Used to denote an integer type.
Char-Used to denote a character type.
float, double-Used to denote a floating point type.
int*,float*,char*- used to denote a pointer type.

Declaration of Primary Data Types with variable name:

After taking suitable variable names, they need to be assigned with a data type. This is how the data types are used along with variables:

Example:

```
int age;
char letter;
float height, width;
```

Derived Data Types

C supports three derived data types:

DATATYPES	DESCRIPTION
Arrays	Arrays are sequences of data items having homogeneous values. They have adjacent memory locations to store values.
References	Function pointers allow referencing functions with a particular signature.
Pointers	These are powerful C features which are used to access the memory and deal with their addresses.

User Defined Data Types

C allows the feature called type definition which allows programmers to define their own identifier that would represent an existing data type. There are three such types:

Data Types	Description
Structure	It is a package of variables of different types under a single name. This is done to handle data efficiently. "struct" keyword is used to define a structure.
Union	These allow storing various data types in the same memory location. Programmers can define a union with different members but only a single member can contain a value at given time.
Enum	Enumeration is a special data type that consists of integral constants and each of them is assigned with a specific name. "enum" keyword is used to define the enumerated data type.

Example for Data Types and Variable Declarations in C

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

int a = 4000; // positive integer data type float b = 5.2324; // float data type
char c = 'Z'; // char data type
long d = 41657; // long positive integer data type long e = -21556; // long -ve integer data type
int f = -185; // -ve integer data type
short g = 130; // short +ve integer data type short h = -130; // short -ve integer data type
double i = 4.1234567890; // double float data type float j = -3.55; // float data type
}
```

Let's see the basic data types. Its size is given according to 32 bit architecture.

Data Types	Memory Size	Range
Char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
Short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
Int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	
double	8 byte	
long double	10 byte	

The storage representation and machine instructions differ from machine to Machine. sizeof operator can use to get the exact size of a type or a variable on a particular platform.

Example:

```
#include <stdio.h>
 #include <limits.h>
int main() {
   printf("Storage size for int is: %d \n", sizeof(int));
   printf("Storage size for char is: %d \n", sizeof(char)); return 0
}
```

CONSTANTS

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc. There are different types of constants in C programming.

List of Constants in C

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in javatpoint" etc

2 ways to define constant in C

There are two ways to define constant in C programming.

1. const keyword
2. #define preprocessor

C const keyword:

The const keyword is used to define constant in C programming.

Example:

```
const float PI=3.14;
```

Now, the value of PI variable can't be changed.

```
#include<stdio.h>
int main()
{
const float PI=3.14;
 printf("The value of PI is: %f",PI);
return 0;
}
```

Output:

The value of PI is: 3.140000

If you try to change the value of PI, it will render compile time error.

```
#include<stdio.h>
int main(){
const float PI=3.14; PI=4.5;
printf("The value of PI is: %f",PI);
return 0;
}
```

Output:

Compile Time Error: Cannot modify a const object.

C #define preprocessor

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax: #define token value

Let's see an example of #define to define a constant.

```
#include <stdio.h>
#define PI 3.14
main()
{
printf("%f",PI);
}
```

Output:

3.140000

Backslash character constant:

C supports some character constants having a backslash in front of it. The lists of backslash characters have a specific meaning which is known to the compiler. They are also termed as “Escape Sequence”.

Example:

\t is used to give a tab

\n is used to give new line

Constants	Meaning	Constants	Meaning
\a	beep sound	\n	newline
\v	vertical tab	\\	backslash

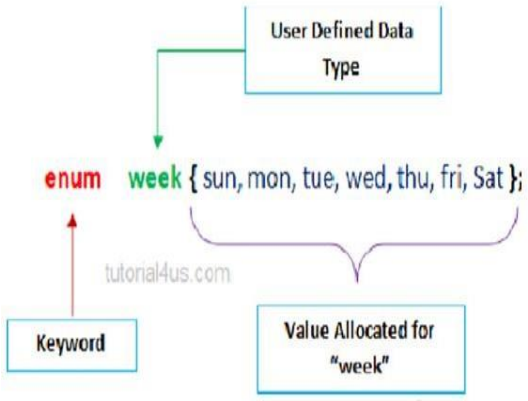
\b	backspace	\r	carriage return
\’	single quote	\0	null
\f	form feed	\t	horizontal tab
\”	double quote		

ENUMERATION CONSTANTS:

An enum is a keyword, it is an user defined data type. All properties of integer are applied on Enumeration data type so size of the enumerator data type is 2 byte . It work like the Integer. It is used for creating an user defined data type of integer. Using enum we can create sequence of integer constant value.

Syntax: enum tagname{value1,value2,value3,….};

- In above syntax enum is a keyword. It is a user defined data type.
- In above syntax tagname is our own variable. tagname is any variable name.
- value1, value2, value3,are create set of enum values.



It is start with 0 (zero) by default and value is incremented by 1 for the sequential identifiers in the list. If constant one value is not initialized then by default sequence will be start from zero and next to generated value should be previous constant value one.

Example:

enum week{sun,mon,tue,wed,thu,fri,sat}; enum week today;

- In above code first line is create user defined data type called week.
- week variable have 7 value which is inside { } braces.

- today variable is declare as week type which can be initialize any data or value among 7 (sun, mon,).

Example:

```
#include<stdio.h>
#include<conio.h>
enum abc{x,y,z};
void main()
{
int a;
 clrscr();
a=x+y+z;        //0+1+2
printf("sum: %d",a);
getch();
}
```

Output:

Sum: 3

KEYWORDS:

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. There are only 32 reserved words (keywords) in C language.

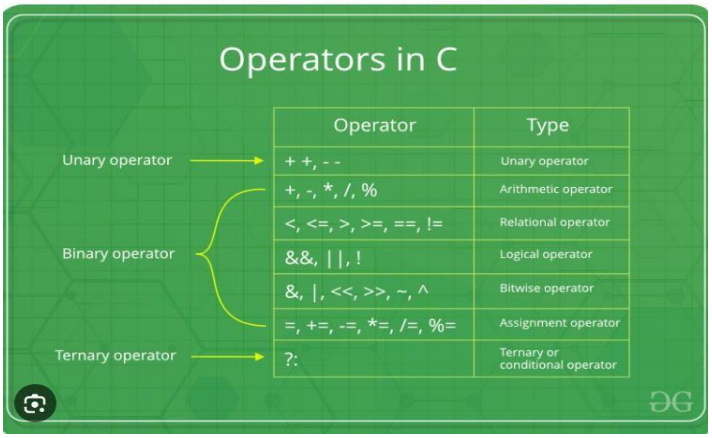
A list of 32 keywords in c language is given below:

auto	break	case	Char	Const	Continue	default	do	Double	else
enum	extern	float	For	Goto	If	int	long	register	return
short	signed	sizeof	Static	Struct	Switch	typedef	union	unsigned	void
volatile	while								

OPERATORS :

Operator is a special symbol that tells the compiler to perform specific mathematical or logical Operation.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Ternary or Conditional Operators



Arithmetic Operators:

Given table shows all the Arithmetic operator supported by C Language. Lets suppose variable A hold 8 and B hold 3.

Operator	Example (int A=8, B=3)	Result
+	A+B	11
-	A-B	5
*	A*B	24
/	A/B	2
%	A%4	0

Relational Operators:

Which can be used to check the Condition, it always return true or false. Lets suppose variable hold 8 and B hold 3.

Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
! =	Not equal to	5! =2	True
===	Equal value and same type	5 === 5	True
		5 === "5"	False
! ==	Not Equal value or Not same type	5 ! == 5	False
		5 ! == "5"	True

Logical Operator:

Which can be used to combine more than one Condition?. Suppose you want to combined two

conditions A<B and B>C, then you need to use Logical Operator like (A<B) && (B>C). Here && is Logical Operator.

Operator	Example (int A=8, B=3, C=-10)	Result
&&	(A<B) && (B>C)	False
||	(B!=-C) || (A==B)	True
!	!(B<=-A)	True

Truth table of Logical Operator

C1	C2	C1&&C2	C1||C2	!C1	!C2
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Assignment operators:

Which can be used to assign a value to a variable. Lets suppose variable A hold 8 and B hold 3.

Operator	Example (int A=8, B=3)	Result
+=	A+=B or A=A+B	11
-=	A-=3 or A=A+3	5
*=	A*=7 or A=A*7	56
/=	A/=B or A=A/B	2
%=	A%=5 or A=A%5	3
a=b	Value of b will be assigned to a	

Increment and Decrement Operator:

Increment Operators are used to increased the value of the variable by one and Decrement Operators are used to decrease the value of the variable by one in C programs.

Both increment and decrement operator are used on a single operand or variable, so it is called as a unary operator. Unary operators are having higher priority than the other operators it means unary operators are executed before other operators.

Increment and decrement operators are cannot apply on constant.

The operators are ++, -- Type of Increment Operator

- pre-increment
- post-increment

pre-increment (++ variable):

In pre-increment first increment the value of variable and then used inside the expression (initialize into another variable).

Syntax:

++variable;

post-increment (variable ++):

In post-increment first value of variable is used in the expression (initialize into another variable) and then increment the value of variable.

Syntax:

variable++;

Example:

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

```
void main()
{
int x,i; i=10;
x=++i;
 printf("Pre-increment\n");
 printf("x::%d",x);
 printf("i::%d",i);
i=10;
x=i++;
 printf("Post-increment\n");
 printf("x::%d",x);
 printf("i::%d",i);
}
```

Output:
Pre-increment x::10
i::10
Post-increment x::10
i::11

Type of Decrement Operator:

- pre-decrement
- post-decrement

Pre-decrement (-- variable):

In pre-decrement first decrement the value of variable and then used inside the expression (initialize into another variable).

Syntax:

--variable;

Post-decrement (variable --):

In Post-decrement first value of variable is used in the expression (initialize into another variable) and then decrement the value of variable.

Syntax:

variable--;

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x,i; i=10;
x=--i;
 printf("Pre-decrement\n");
 printf("x::%d",x);
 printf("i::%d",i);
i=10;
x=i--;
 printf("Post-decrement\n");
 printf("x::%d",x);
 printf("i::%d",i);
}
```

Output:
Pre-decrement x::9
i::9
Post-decrement x::10
i::9

Ternary Operator:

If any operator is used on three operands or variable is known as Ternary Operator. It can be represented with ? : . It is also called as conditional operator

Advantage of Ternary Operator

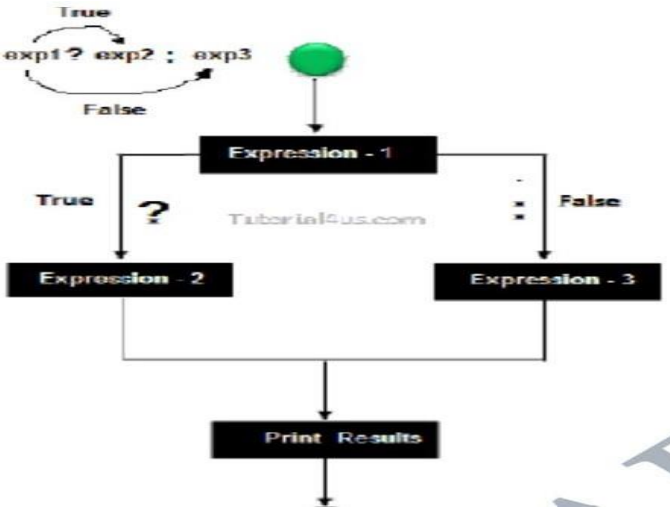
Using ?: reduce the number of line codes and improve the performance of application.

Syntax:

Expression 1? Expression 2: Expression 3;

In the above symbol expression-1 is condition and expression-2 and expression-3 will be either value Or variable or statement or any mathematical expression. If condition will be true expression-2 will be execute otherwise expression-3 will be executed.

Conditional Operator flow diagram



Example:

find largest number among 3 numbers using ternary operator

```
#include<stdio.h>
void main()
{
int a,b,c,large;
printf("Enter any three numbers:");
scanf("%d%d%d",&a,&b,&c);
large=a>b?(a>c?a:c):(b>c?b:c);
printf("The largest number is:%d",large);
}
```

Output:
Enter any three numbers: 12 67 98
The largest number is 98

Special Operators:

C supports some special operators

Operator	Description
sizeof()	Returns the size of an memory location.
&	Returns the address of an memory location.
*	Pointer to a variable.

Expression evaluation

In C language expression evaluation is mainly depends on priority and associativity.

Priority

This represents the evaluation of expression starts from "what" operator.

Associativity

It represents which operator should be evaluated first if an expression is containing more than one operator with same priority.

Precedence of operators :

The precedence rule is used to determine the order of application of operators in evaluating sub expressions. Each operator in C has a precedence associated with it. The operator with the highest precedence is operated first.

Associativity of operators :

The associativity rule is applied when two or more operators are having same precedence in the sub expression. An operator can be left-to-right associative or right-to-left associative.

Rules for evaluation of expression:

- First parenthesized sub expressions are evaluated first.
- If parentheses are nested, the evaluation begins with the innermost sub expression.
- The precedence rule is applied to determine the order of application of operators in evaluating

sub expressions.

- The associability rule is applied when two or more operators are having same precedence in the sub expression.

Operator	*Description*	*Associativity*
() **[]** **.** **->** **++ --**	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- **+ -** **! ~** **(type)** ***** **&** **sizeof**	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
*** / %**	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= **> >=**	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to and not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= **+= -=** ***= /=** **%= &=** **^= |=** **<<= >>=**	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	Comma operator	left to right

EXPRESSION:

An expression is a sequence of operators and operands that specifies computation of a value.

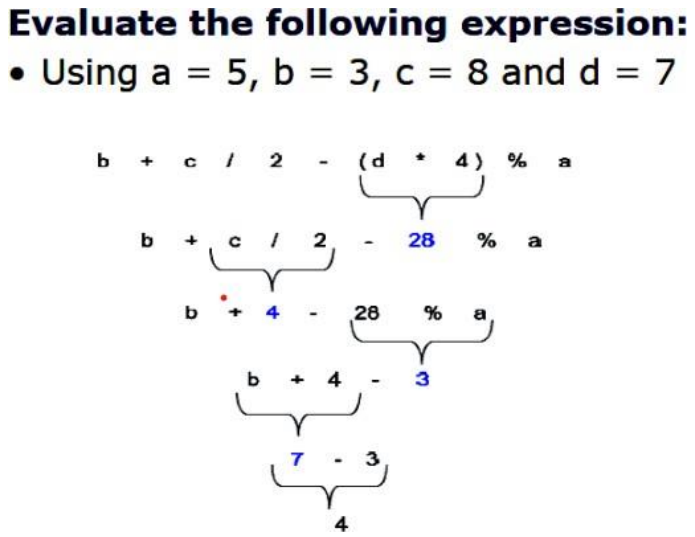
For e.g, a=2+3 is an expression with three operands a,2,3 and 2 operators = & +

Simple Expressions & Compound Expressions:

An expression that has only one operator is known as a simple expression. E.g: a+2

An expression that involves more than one operator is called a compound expression.

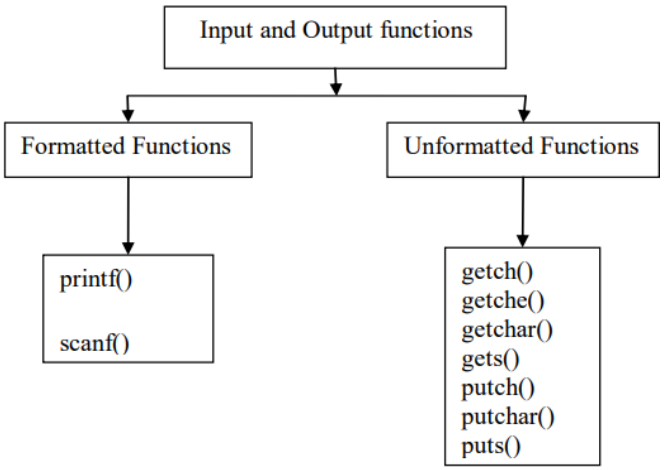
E.g: b=2+3*5.



IO STATEMENT:

The I/O functions are classified into two types:

- Formatted Functions
- Unformatted functions



FORMATTED INPUT FUNCTION:

SCANF():

It is used to get data in a specified format. It can accept different data types.

Syntax:

scanf("Control String", var1address, var2address, …);

Data Types	Format Specifier
int	%d
short	%d
long	%ld
char	%c
float	%f
double	%lf
long double	%Lf

EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    int a,b,sum;
    clrscr();
    scanf("%d %d",&a,&b);
    sum= a+b;
}
```

FORMATTED OUTPUT FUNCTION:

PRINTF():

The printf( ) function is used to print data of different data types on the console in a specified format.

Syntax:

printf("Control String", var1, var2, …);

EXAMPLE:

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

```
{
    int a,b,sum;
    clrscr();
    printf("enter two numbers:");
    scanf("%d %d",&a,&b);
    sum= a+b;
printf("sum is:%d",sum);
}
```

```
Enter two numbers: 5    4
Sum is 9
```

UNFORMATTED INPUT FUNCTION:

- getchar()
- getch()
- getche()
- gets()

getchar():

This function reads a single character data from the standard input.

Syntax:

variable_name=getchar();

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
Char ch;
ch=getchar();
Printf("%c",ch);
}
```

OUTPUT:
j

getch():

getch() accepts only a single character from keyboard. The character entered through getch() is not displayed in the screen (monitor).

Syntax:

variable_name = getch();

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
Char ch;
ch=getch();
Printf("ch=%c",ch);
}
```

OUTPUT:
Ch=a

getche():

getche() also accepts only single character, but getche() displays the entered character in the screen.

Syntax:

variable_name = getche();

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
Char ch;
ch=getche();
Printf("ch=%c",ch);
}
```

OUTPUT:
a
Ch=a

gets():

This function is used for accepting any string through stdin (keyboard) until enter key is pressed.

Syntax:

gets(variable_name);

Example:

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

```
void main()
{
Char ch[10];
gets(ch);
Printf("ch=%s",ch);
getch();
}
```

OUTPUT:
cprogram
Ch=cprogram

UNFORMATTED OUTPUT FUNCTION:

- putchar()
- putch()
- puts()

putchar():

This function prints one character on the screen at a time.

Syntax :

putchar(variable name);

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
Char ch;
printf("enter a character:");
ch=getchar();
putchar(ch);
getch();
}
```

OUTPUT:
enter a character: j
j

putch():

putch displays any alphanumeric characters to the standard output device. It displays only one character at a time.

Syntax:

putch(variable_name);

Example:

```
include<stdio.h>
#include<conio.h>
void main()
{
  char ch;
  clrscr();
  printf("Press any character: ");
  ch = getch();
  printf("\nPressed character is:");
  putch(ch);
  getch();
}
```

OUTPUT:
Press any character:
Pressed character is: e

puts():

This function prints the string or character array.

Syntax:

puts(variable_name);

Example:

```
include<stdio.h>
#include<conio.h>
void main()
{
  char ch[20];
  clrscr();
puts("enter a string");
gets(ch);
puts(ch);
}
```

OUTPUT:
Enter a string: cprogramming
cprogramming

ASSIGNMENT STATEMENT:

The assignment statement has the following form:

Syntax:

variable = expression/constant/variable;

Its purpose is saving the result of the expression to the right of the assignment operator to the variable on the left. Here are some rules:

- If the type of the expression is identical to that of the variable, the result is saved in the variable.
- Otherwise, the result is converted to the type of the variable and saved there.
    - ❖ If the type of the variable is integer while the type of the result is real, the fractional part, including the decimal point, is removed making it an integer result.
    - ❖ If the type of the variable is real while the type of the result is integer, then a decimal point is appended to the integer making it a real number.
- Once the variable receives a new value, the original one disappears and is no more available.

Examples of assignment statements,

```
b = c ; /* b is assigned the value of c */

a = 9 ; /* a is assigned the value 9*/

b = c+5; /* b is assigned the value of expr c+5 */
```

- The expression on the right hand side of the assignment statement can be: An arithmetic expression;
    - ❖ A relational expression;
    - ❖ A logical expression;
    - ❖ A mixed expression.

For example,

```
int a;

float b,c ,avg, t;

avg = (b+c) / 2; /*arithmetic expression */

a = b && c; /*logical expression*/

a = (b+c) && (b<c); /* mixed expression*/
```

DECISION MAKING STATEMENTS:

Decision making statement is depending on the condition block need to be executed or not which is decided by condition.

If the condition is "true" statement block will be executed, if condition is "false" then statement block will not be executed.

In this section we are discuss about if-then (if), if-then-else (if else), and switch statement. In C language there are three types of decision making statement.

- if
- if-else
- switch

if Statement:

if-then is most basic statement of Decision making statement. It tells to program to execute a certain part of code only if particular condition is true.

Syntax:

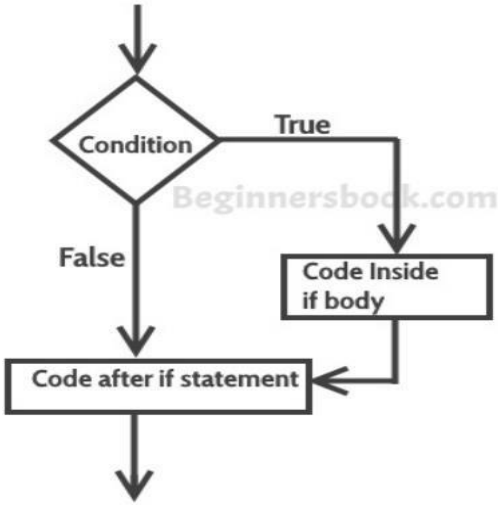
if(condition)

{

Statements executed if the condition is true

}

FLOWCHART:



Constructing the body of "if" statement is always optional, Create the body when we are having multiple statements.

For a single statement, it is not required to specify the body.

If the body is not specified, then automatically condition part will be terminated with next semicolon (;).

Example:

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

```
        printf("enter a number:");
scanf("%d",&num);
        if(num%2==0)
        {
        printf("%d is even number",num);
        }
        return 0;
        }
```

OUTPUT:
Enter a number: 4
4 is even number

if-else statement:

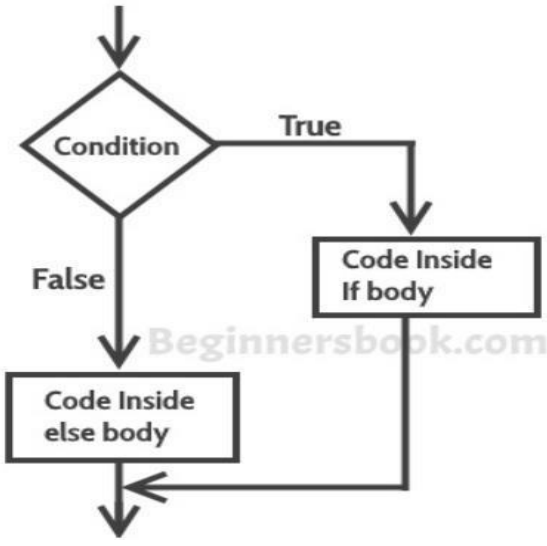
In general it can be used to execute one block of statement among two blocks, in C language if and else are the keyword in C.

Syntax:

```
if(expression)
    {


      else
```

Flowchart:



In the above syntax whenever condition is true all the if block statement are executed remaining statement of the program by neglecting else block statement. If the condition is false else block statement remaining statement of the program are executed by neglecting if block statements.

Example:

```
#include<stdio.h>
void main()
{
int age;
printf("enter age:")
scanf("%d",&age);
if(age>=18)
{
printf("age:%d",age);
printf("eligible to vote" );
}
else
{
printf("age:%d",age);
printf("not eligible to vote" );
}
```

Output:
Enter age: 18
Eligible to vote
Enter age: 17
Not eligible to vote

Nested if:

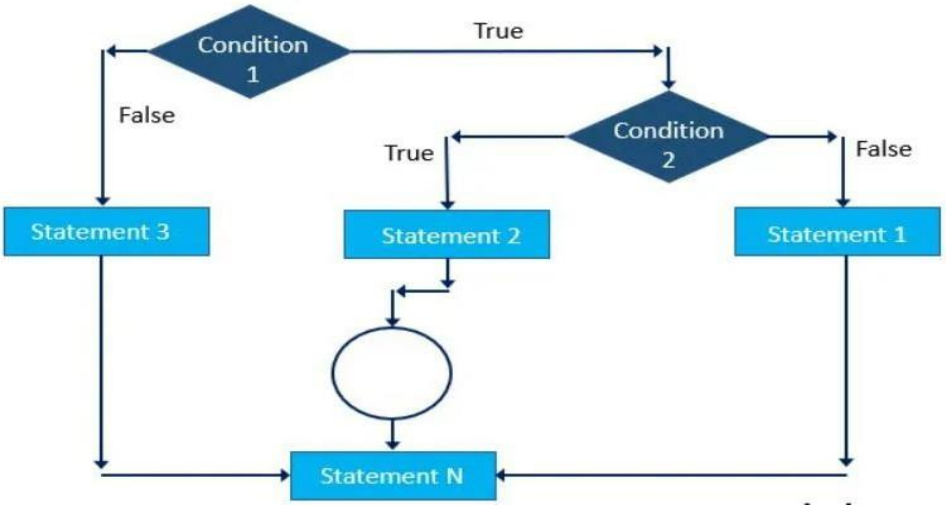
When an if else statement is present inside the body of another "if" or "else" then this is called nested if else.

Syntax of Nested if else statement:

```
if(condition) {
   //Nested if else inside the body of "if"
   if(condition2) {
      //Statements inside the body of nested "if"
   }
   else {
      //Statements inside the body of nested "else"
   }
}
else {
   //Statements inside the body of "else"
```

}

Flowchart:



EXAMPLE:

```
#include<stdio.h>
void main()
{
int age, salary;
printf("enter age and salary");
scanf(%d %d", &age,&salary);
if(age>50)
{
   if(salary<60000)
     {
          salary=salary+10000
          ;printf("%d",salary);
       }
    else
       {
          salary=
          salary+5000;
          printf("%d",salary);
```

Output:
Enter age and salary: 55 55000
65000

```
        }
    }
    else
    {
    salary=salary+1000;
    printf("%d",salary);
    }
    printf("end of program");
    getch();
    }
```

Switch:

A switch statement work with byte, short, char and int primitive data type, it also works with enumerated types and string.

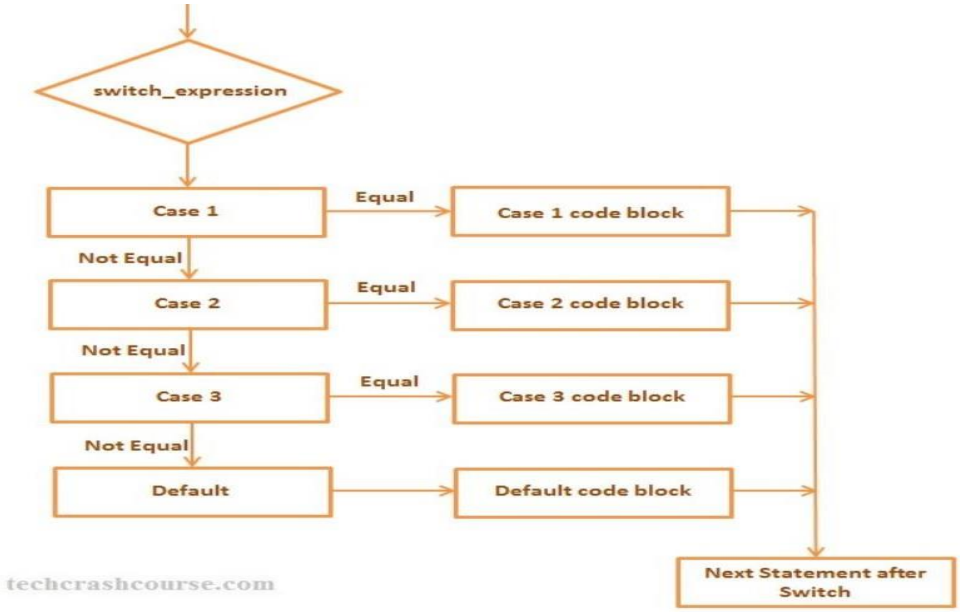
Syntax:

```
switch(expression/variable)
{
        case value1:
        statements;
        break;//optional
        case value2:
        statements;
        break;//optional
        default:
                statements;
        break;//optional
}
```

Rules for apply switch:

1. With switch statement use only byte, short, int, char data type.
2. You can use any number of case statements within a switch.
3. Value for a case must be same as the variable in switch

Flowchart:



Example:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
void main()
{
   // declaration of local variable op;
   int op, n1, n2;
          printf (" enter 2 number: ");
          scanf(%d %d",&n1,&n2);
      printf (" \n 1 Addition  \t \t 2 Subtraction \n 3 Multiplication \t 4 Division \n 5 Exit \n \n Please, Make a choice ");
       scanf ("%d", &op); // accepts a numeric input to choose the operation
   switch (op)
   {
      case 1:
         printf ("sum is :%d ",n1+n2);
         break;
```

```
case 2:
   printf ("difference is :%d ",n1-n2);
   break;
case 3:
   printf ("multiplication :%d ",n1*n2);
   break;
case 4:
   printf ("division :%d ",n1/n2);
   break;
case 5:
   printf ("exit");
   break;
   default:
        printf("enter the number between 1 to 5:");
   }
   }
```

LOOPING STATEMENTS

Sometimes it is necessary for the program to execute the statement several times, and C loops execute a block of commands a specified number of times until a condition is met.

What is Loop?

A computer is the most suitable machine to perform repetitive tasks and can tirelessly do a task tens of thousands of times. Every programming language has the feature to instruct to do such repetitive tasks with the help of certain form of statements. The process of repeatedly executing a collection of statement is called looping . The statements get executed many numbers of times based on the condition. But if the condition is given in such a logic that the repetition continues any number of times with no fixed condition to stop looping those statements, then this type of looping is called infinite looping.

C supports following types of loops:

- while loops
- do while loops
- for loops

while loops:

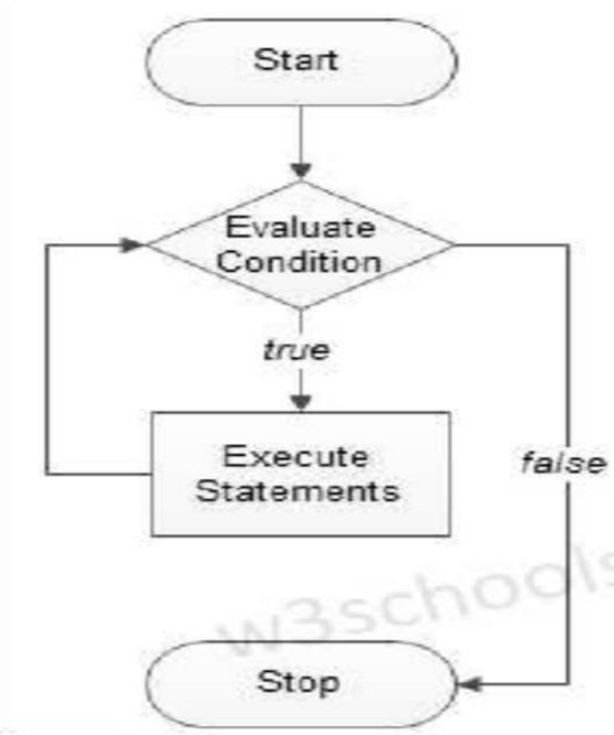
C while loops statement allows to repeatedly run the same block of code until a condition is met. while loop is a most basic loop in C programming. while loop has one control condition, and executes as long the condition is true.

The condition of the loop is tested before the body of the loop is executed, hence it is called an entry-controlled loop.

Syntax:

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

Flowchart:



Flowchart:

Example: #include<stdio.h>void main ()

```
{
int i=1;
clrscr();
while(i<=10)
{
printf("%d",i);
i++;
} getch();
}
```

Output:
1 2 3 4 5 6 7 8 9 10

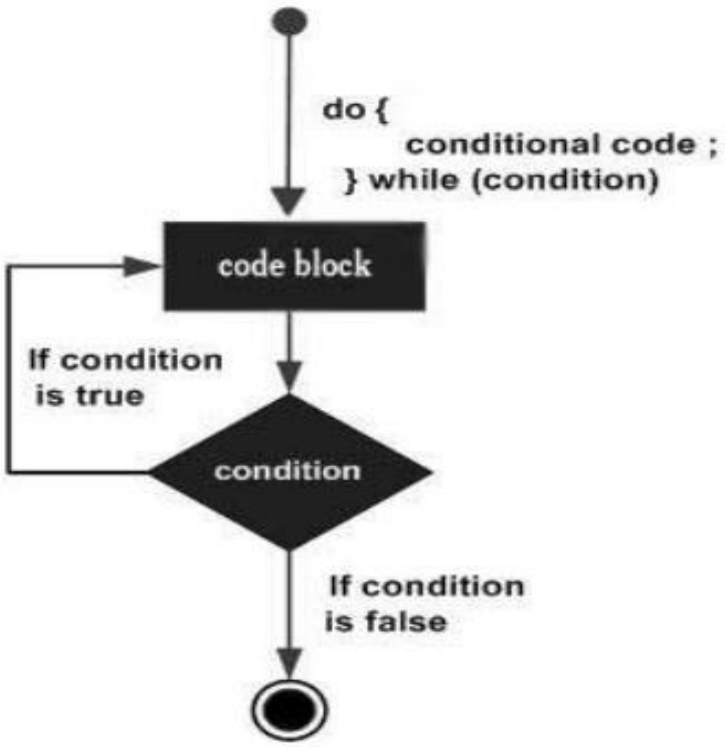
Do..while loops:

C do while loops are very similar to the while loops, but it always executes the code block at least once and furthermore as long as the condition remains true. This is an exit- controlled loop.

Syntax:

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

Flowchart:


do {
conditional code ;
} while (condition)
code block
If condition
is true
condition
If condition
is false

EXAMPLE:

```
#include<stdio.h>
void main ()
{
int  i=1;
clrscr();
do
{
printf("%d",i);
i++;
} while(i<=10);
 getch();
}
```

Output:
1 2 3 4 5 6 7 8 9 10

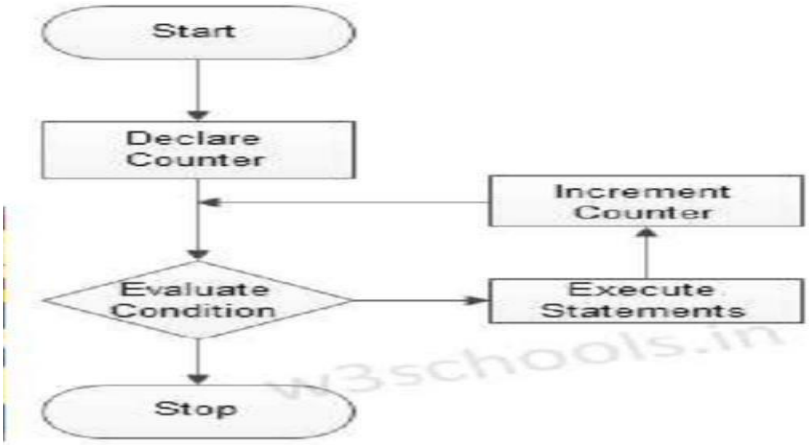
For loop:

C for loops is very similar to a while loops in that it continues to process a block of code until a statement becomes false, and everything is defined in a single line. The for loop is also entry-controlled loop.

Syntax:

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

Flowchart:



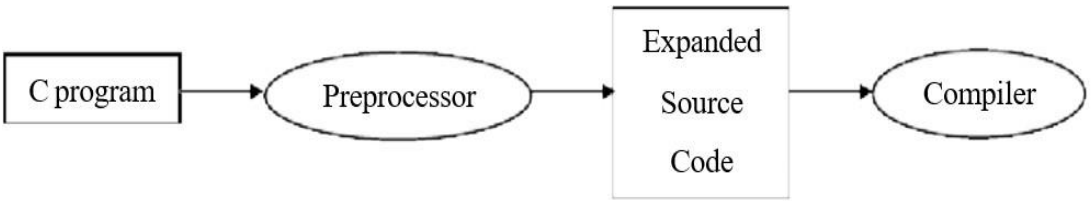
Example:

```
#include<stdio.h>
void main ()
{
int i;
clrscr();
for(i=1;i<=10;i++)
{
printf("%d",i);
getch();
}
```

Output:
1 2 3 4 5 6 7 8 9 10

PRE-PROCESSOR DIRECTIVES

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros. Preprocessor directives are executed before compilation.



All preprocessor directives starts with hash #symbol.

Let's see a list of preprocessor directives.

- #include
- #define

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

s.no	Preprocessor directive	purposes	syntax
1	#include	Used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.	#include <filename> #include “filename”
2	#define	Used to define constant or microsubstitution. It can use any basic data type.	#define PI 3.14
3	#undef	Used to undefine the constant or macro defined by #define.	#undef PI
4	#ifdef	Checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.	#ifdef MACRO //code #endif
5	#ifndef	Checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.	#ifndef MACRO //code #endif
6	#if	Evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed	#if expression //code #endif

7	#else	Evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.	#if expression
			//if code
			#else
			//else code
			#endif

8	#error	Indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.	#error First include then compile
9	#pragma	Used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature.	#pragma token

COMPILATION PROCESS:

C is a high level language and it needs a compiler to convert it into an executable code so that the program can be run on our machine.

How do we compile and run a C program?

Below are the steps we use on an Ubuntu machine with gcc compiler.

- We first create a C program using an editor and save the file as filename.c
  $ vi filename.c
  The diagram on right shows a simple program to add two numbers.

compile it using below command.

$ gcc –Wall filename.c –o filename

The option -Wall enables all compiler's warning messages. This option is recommended to generate bettercode. The option -o is used to specify output file name. If we do not use this option, then an output file with name a.out is generated.

After compilation executable is generated and we run the generated executable using below command.

$ ./filename

What goes inside the compilation process?

Compiler converts a C program into an executable. There are four phases for a C program to become an executable:

1. Pre-processing
2. Compilation
3. Assembly
4. Linking

By executing below command, We get the all intermediate files in the current directory along with the executable.

$gcc –Wall –save-temps filename.c –o filename

The following screenshot shows all generated intermediate files.

Let us one by one see what these intermediate files contain

Pre-processing:

This is the first phase through which source code is passed. This phase include:

- Removal of Comments
- Expansion of Macros
- Expansion of the included files.

The preprocessed output is stored in the filename.i. Let's see what's inside

filename.i:using $vi filename.i

In the above output, source file is filled with lots and lots of info, but at the end our code is preserved.

Analysis:

- printf contains now a + b rather than add(a, b) that's because macros have expanded.
- Comments are stripped off.

- #include<stdio.h> is missing instead we see lots of code. So header files has been expanded andincluded in our source file.

Compiling:

The next step is to compile filename.i and produce an; intermediate compiled output file filename.s.

This file is in assembly level instructions. Let's see through this file using $vi filename.s

Assembly:

In this phase the filename.s is taken as input and turned into filename.o by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved. Let's view this file using $vi filename.o

Linking:

This is the final phase in which all the linking of function calls with their definitions are done. Linkerknows where all these functions are implemented.

Linker does some extra work also, it adds some extra code to our program which is required whenthe program starts and ends.

For example, there is a code which is required for setting up the environment like passing commandline arguments. This task can be easily verified by using $size filename.o and $size filename.

Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.

UNIT II

ARRAYS AND STRINGS

Introduction to Arrays: Declaration, Initialization – One dimensional array – Two dimensional arrays - String operations: length, compare, concatenate, copy – Selection sort, linear and binary search.

INTRODUCTION TO ARRAYS: DECLARATION,INITIALIZATION:ONE

DIMENSIONAL ARRAYS:

Array in C language is a collection or group of elements (data). All the elements of c array are homogeneous (similar). It has contiguous memory location.

C array is beneficial if you have to store similar elements. Suppose you have to store marks of 50 students, one way to do this is allotting 50 variables.

So it will be typical and hard to manage.

For example we cannot access the value of these variables with only 1 or 2 lines of code.

Another way to do this is array. By using array, we can access the elements easily. Only few lines of code is required to access the elements of array.

Advantage of C Array:

1) Code Optimization: Less code to the access the data.

2) Easy to traverse data: By using the for loop, we can retrieve the elements of an array easily.

3) Easy to sort data: To sort the elements of array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array:

1) Fixed Size: Whatever size, we define at the time of declaration of array, we can't exceed the limit. So, it doesn't grow the size dynamically like Linked List.

Declaration of C Array:

We can declare an array in the c language in the following way.

Syntax:

data_type array_name[array_size];

Now, let us see the example to declare array.

int marks[5];

Here**, int** is the datatype**, marks** is the array_name and **5** is the array_size.

How to access element of an array in C:

You can use **array subscript** (or index) to access any element stored in array. Subscript starts with 0, which means arr[0] represents the first element in the array arr.

In general arr[n-1] can be used to access nth element of an array. where n is any integer number.

Initialization of C Array:

A simple way to initialize array is by index. Notice that array index starts from 0 and ends with [SIZE - 1].

```
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
```

Example:

```
#include<stdio.h>
 int main()
 {
 int i=0;
 int marks[5];//declaration of array
 marks[0]=80;//initialization of array
 marks[1]=60;
 marks[2]=70;
 marks[3]=85;
 marks[4]=75;
 //traversal of array
 for(i=0;i<5;i++)
 {
 printf("%d \n",marks[i]);
 }//end of for loop return 0;
 }
```

Output:
80
60
70
85
75

C Array: Declaration with Initialization:

We can initialize the c array at the time of declaration. Let's see the code. int marks[5]={20,30,40,50,60};

In such case, there is no requirement to define size. So it can also be written as the following code

.

Example: int marks[]={20,30,40,50,60};

Programs:

```
#include<stdio.h>
int main()
{
int i=0;
int marks[5]={20,30,40,50,60};//declaration and initialization of array
//traversal of array
for(i=0;i<5;i++)
{
 printf("%d \n",marks[i]);
}
return 0;
}
```

Output:
20
30
40
50
60

TWO DIMENSIONAL ARRAYS (2 D arrays):

The two dimensional array in C language is represented in the form of rows and columns, also known as matrix. It is also known as array of arrays or list of arrays.

The two dimensional, three dimensional or other dimensional arrays are also known as multidimensional arrays.

Declaration of two dimensional Array in C:

We can declare an array in the c language in the following way.

Syntax:

data_type array_name[size1][size2];

A simple example to declare two dimensional array is given below.

Example: int twodimen[4][3];

Here, 4 is the row number and 3 is the column number.

Initialization of 2D Array in C:

A way to initialize the two dimensional array at the time of declaration is given below.

Example: int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};

Programs:

```
#include<stdio.h>
int main()
{
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
 for(i=0;i<4;i++)
{
 for(j=0;j<3;j++)
{
 printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
}//end of j
}//end of i return 0;
}
```

Output:

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

STRING OPERATION:

What is meant by String?

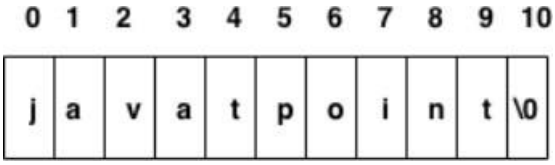
String in C language is an array of characters that is terminated by \0 (null character). There are two ways to declare string in c language.

1. By char array
2. By string literal

Let's see the example of declaring string by char array in C language.

```
char ch[10]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```

As you know well, array index starts from 0, so it will be represented as in the figure given below.



While declaring string, size is not mandatory. So you can write the above code as given below:

char ch[]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

You can also define string by string literal in C language.

For example: char ch[]="javatpoint";

In such case, '\0' will be appended at the end of string by the compiler.

Difference between char array and string literal:

The only difference is that string literal cannot be changed whereas string declared by char array can be changed.

Programs:

Let's see a simple example to declare and print string. The '%s' is used to print string in c language**.**

```
#include<stdio.h>
#include <string.h>
int main()
{
char ch[11]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
char ch2[11]="javatpoint";
printf("Char Array Value is: %s\n", ch);
printf("String Literal Value is: %s\n", ch2);
return 0;
}
```

Output:

```
Char Array Value is: javatpoint
String Literal Value is: javatpoint
```

1. String operations: length-strlen()

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

Example:

```
#include<stdio.h>
 #include <string.h>
 int main()
{
char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
 printf("Length of string is: %d",strlen(ch));
return 0;
}
```

Output:

```
Length of string is: 10
```

2. String operations: compare-strcmp():

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using gets() function which reads string from the console.

Programs:

```
#include<stdio.h>
#include <string.h>
int main()
{
char str1[20],str2[20];
 printf("Enter 1st string: ");
 gets(str1);//reads string from console
printf("Enter 2nd string: ");
gets(str2);
if( (strcmp(str1,str2)==0) printf("Strings
    are equal");
else
 printf("Strings are not equal");
return 0;
}
```

Output:
Enter 1st string: hello
Enter 2nd string: hello
Strings are equal

3. String operations: concatenate-strcat():

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

Programs:

```
#include<stdio.h>
#include <string.h>
int main()
{
    char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
    char ch2[10]={'c', '\0'};
    strcat(ch,ch2);
printf("Value of first string is: %s",ch);return
0;
}
```

Output:
Value of first string is: helloc

4. String operations: copy-strcpy():

The strcpy(destination, source) function copies the source string in destination

Programs:

```
#include<stdio.h>
 #include <string.h>
 int main()
{
char ch[20]={'j', 'a', 'v', 'a', 't',  'p', 'o', 'i', 'n', 't', '\0'};
 char ch2[20];
strcpy(ch2,ch);
 printf("Value of second string is: %s",ch2);
return 0;
}
```

Output:
Value of second string is: javatpoint

5. String operations:Reverse - strrev():

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

Programs:

```
#include<stdio.h>
#include <string.h>int
main()
{
        char str[20];
        printf("Enter string: ");
        gets(str);//reads string from console
        printf("String is: %s",str);
        printf("\nReverse String is: %s",strrev(str));
       return 0;
}
```

Output:

Enter string: javatpoint

String is: javatpoint

Reverse String is: tnioptavaj

6. String operation: lower- strlwr():

The strlwr(string) function returns string characters in lowercase. Let's see asimple example of strlwr() function.

Programs:

```
#include<stdio.h>
#include <string.h>int
main()
{
        char str[20];
        printf("Enter string: ");
        gets(str);//reads string from console
        printf("String is: %s",str);
        printf("\nLower String is: %s",strlwr(str));
       return 0;
}
```

Output:

Enter string: JAVATpoint
String is: JAVATpoint
Lower String is: javatpoint

7. String operation:upper-strupr():

The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() function.

Programs:

```
#include<stdio.h>
#include <string.h>
int main()
{
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nUpper String is: %s",strupr(str));
 return 0;
}
```

Output:

Enter string: javatpoint
String is: javatpoint
Upper String is: JAVATPOINT

Sample Programs:

1. **Program to calculate the average marks of the class**

```
#include<stdio.h>void
main()
{ int m[5],i,sum=0,n;float
avg;
printf(“enter number of students \n”);
scanf(“%d”,&n);
printf(“enter marks of students \n”);
for(i=0;i <n;i++)
sum=sum+m[i]; avg=float(sum)/n;
printf(“average of:%f”,avg);
}
```

Output:

```
Enter number of students
        5
Enter marks of students
        55
        60
        78
        85
        90
Average of: 73.6
```

2. **Addition of two numbers in array:**

```
#include <stdio.h>
void main()
{
    int a[10], b[10], c[10], n, i;
   printf("Enter the number of elements:\t");
    scanf("%d", &n);
    printf("Enter %d elements for array 1:\n", n);
   for (i = 0; i < n; i++)
       scanf("%d", &a[i]);
    printf("Enter %d elements for array 2:\n", n);
   for (i = 0; i < n; i++)
       scanf("%d", &b[i]);
   for (i = 0; i < n; i++)
       c[i] = a[i] + b[i];
  printf("Sum of two array elements are:\n");
  for (i = 0; i < n; i++)
       printf("%d\n", c[i]);
}
```

Output:

```
Enter the number of elements: 5Enter 5
elements for array 1:
93
37
71
03
17
Enter 5 elements for array 2:
29
84
28
75
63
Sum of two array elements are:122
```

121
99
78
80

3. **Subtraction of two number:**

```
#include < stdio.h >
int main()
{
        int m, n, c, d, first[10][10], second[10][10], difference[10][10];
        printf("Enter the number of rows and columns of matrix\n");
        scanf("%d%d", & m, & n);
        printf("Enter the elements of first matrix\n");
        for (c = 0; c < m; c++)
           for (d = 0; d < n; d++) scanf("%d", & first[c][d]);
        printf("Enter the elements of second matrix\n");
        for (c = 0; c < m; c++)
           for (d = 0; d < n; d++) scanf("%d", & second[c][d]);
        printf("Difference of entered matrices:-\n");
        for (c = 0; c < m; c++)
        {
           for (d = 0; d < n; d++)
           {
              difference[c][d] = first[c][d] - second[c][d];
              printf("%d\t", difference[c][d]);
           }
           printf("\n");
        }
        return 0;
}
```

Output

```
Enter the number of rows and columns of matrix
2
2
Enter the elements of first matrix
5 4
2 1
Enter the elements of second matrix
4 3
1 2
Difference of entered matrices:-
1       1
1       -1
Enter the number of rows and columns of matrix
```

4. Multiplication of two numbers in array:

```
#include<stdio.h>
#include<stdlib.h>    int
main(){
int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
system("cls");
printf("enter the number of row=");
scanf("%d",&r);
printf("enter the number of column=");
scanf("%d",&c);
printf("enter the first matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("enter the second matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&b[i][j]);
}
}

printf("multiply of the matrix=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
```

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

Output:

```
enter the number of row=3
enter the number of column=3
enter the first matrix element=
1 1 1
2 2 2
3 3 3
enter the second matrix element=
1 1 1
2 2 2
3 3 3
multiply of the matrix=
6 6 6
12 12 12
18 18 18
```

UNIT III FUNCTIONS AND POINTERS

Introduction to functions: Function prototype, function definition, function call, Built-in functions (string functions, math functions) – Recursion – Example Program: Computation of Sine series, Scientific calculator using built-in functions, Binary Search using recursive functions – Pointers – Pointer operators – Pointer arithmetic – Arrays and pointers – Array of pointers – Example Program: Sorting of names – Parameter passing: Pass by value, Pass by reference – Example Program: Swapping of two numbers and changing the value of a variable using pass by reference.

FUNCTIONS

Definition

C enables its programmers to break up a program into segments commonly known as functions

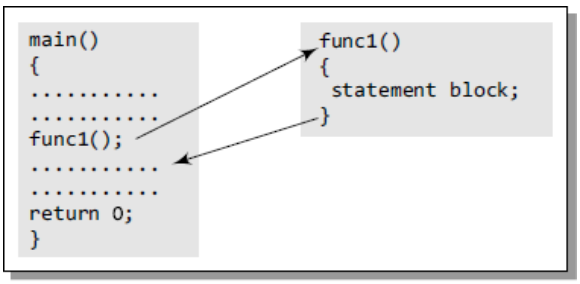


Figure 1.9 `main()` calls `func1()`

Every function in the program is supposed to perform a well-defined task. Therefore, the programcode of one function is completely insulated from the other functions.

main() calls a function named func1(). Therefore, main() is known as the calling function andfunc1() is known as the called function.

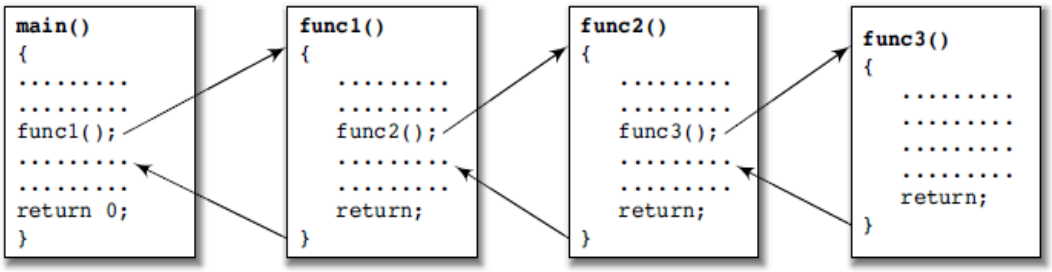


Figure 1.10 Function calling another function

Need For Functions:

Functions are used because of following reasons –

a) To improve the readability of code.

b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.

c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.

d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Terminologies In Functions

- A function f that uses another function g is known as the **calling function**, and g is known as the called function.
- The inputs that a function takes are known as **arguments.**
- When a called function returns some result back to the calling function, it is said to returnthat result.
- The calling function may or may not pass parameters to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
  - **Function declaration** is a declaration statement that identifies a function's name, a list ofarguments that it accepts, and the type of data it returns.
  - **Function definition** consists of a function header that identifies the function, followed by the bodyof the function containing the executable code for that function.

Function Declaration

The general format for declaring a function that accepts arguments and returns a value as result can be given as:

```
return_data_type  function_name(data_type variable1, data_type variable2,..);
```

- function_name - is a valid name for the function. Naming a function follows the same rules that are followed while naming variables. A function should have a meaningful name that must specify the task that the function will perform.
- return_data_type - the data type of the value that will be returned to the calling function asa result of the processing performed by the called function.
- (data_type variable1, data_type variable2, ...) - is a list of variables of specified data types. These variables are passed from the calling function to the called function. They are also known as arguments or parameters that the called function accepts to perform its task.

Function Definition

When a function is defined, space is allocated for that function in the memory. A functiondefinition comprises of two parts:

- Function header
- Function body

The syntax of a function definition can be given as:

```
return_data_type  function_name(data_type variable1, data_type variable2,..)
        {
                .............
                statements
                .............
                return(variable);
        }
```

return_data_type function_name(data_type variable1, data_type variable2,...) is known as the function header, the rest of the portion comprising of program statements within the curly brackets { } is thefunction body which contains the code to perform the specific task.

- The number of arguments and the order of arguments in the function header must be the same as thatgiven in the function declaration statement.
- The function header is same as the function declaration. The only difference between the two is thata function header is not followed by a semi-colon.

Function Call

The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function. A function call statement has the following syntax:

```
function_name(variable1, variable2, ...);
```

If the return type of the function is not void, then the value returned by the called functionmay be assigned to some variable as given below.

```
variable_name = function_name(variable1, variable2, ...);
```

Eg:// program to find whether a number is even or odd using functions.

```
#include <stdio.h>
int evenodd(int);  //FUNCTION DECLARATION
int main()
{
        int num, flag;
        printf("\n Enter the number : ");
        scanf("%d",&num);
        flag = evenodd(num); //FUNCTION CALL
        if (flag == 1)
                printf("\n %d is EVEN", num);
        else
                printf("\n %d is ODD", num);
        return 0;
}
int evenodd(int a) // FUNCTION HEADER
{

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

Output:

Enter the number : 7878 is EVEN

PASSING PARAMETERS TO FUNCTIONS

There are two ways in which arguments or parameters can be passed to the called function.

1. Call by value - The values of the variables are passed by the calling function to the calledfunction.
2. Call by reference - The addresses of the variables are passed by the calling function to thecalled function.

1. Call by Value

- In call by value method, the value of the actual parameters is copied into the formal parameters.
- In call by value method, we cannot modify the value of the actual parameter by the formalparameter.
- In call by value, different memory is allocated for actual and formal parameters.
- The actual parameter is the argument which is used in the function call whereas formal parameter isthe argument which is used in the function definition.

Eg://Program for call by value

```
#include<stdio.h>
int main()
{
        int x,y;
        void swap(int,int); printf("Enter two
        numbers:");scanf("%d%d",&x,&y);
        printf("\n\nBefore Swapping: x = %d\ty =
        %d",x,y);
        swap(x,y);
        printf("\n\nAfter Swapping: x = %d\ty = %d",x,y);
}
void swap(int a, int b)
{
         a=a+b;
        b=a-b;
         a=a-b;
  printf("\n\nIn swap function: x = %d\ty = %d\n\n",a,b);
}
```

Output:

```
Enter two numbers: 2 3
Before Swapping: x=2 y=3
In Swap function: x=3 y=2
After Swapping : x=2 y=3
```

Pros and cons

- The biggest advantage of using the call-by-value technique is that arguments can be passed as variables, literals, or expressions.

1. Its main drawback is that copying data consumes additional storage space. In addition, it can take alot of time to copy, thereby resulting in performance penalty, especially if the function is called many times.

Call By Reference:

- The method of passing arguments by address or reference is also known as call by address or call by reference. Here, the addresses of the actual arguments are passed to the formal parameters of the function.
- If the arguments are passed by reference, changes in the formal parameters also make changes on actual parameters.

Eg//Program for Call by reference

```
#include<stdio.h>
int main()
{
 int x ,y;
void swap (int*,int*);
printf(“enter the two numbers”);
scanf(“%d%d”,&x,&y);
printf("\n\nBefore Swapping:\n\nx = %d\ty = %d",x,y);
swap(&x,&y);
printf("\n\nAfter Swapping:\n\nx = %d\ty = %d\n\n",x,y);
}
void swap(int *a, int *b)
{
                *a=*a+*b;
                *b=*a-*b;
                *a=*a-*b;
                printf("\n\nIn swap function: x = %d\ty = %d\n\n",a,b);
}
```

Output:

```
Enter two numbers: 2 3
Before Swapping: x=2            y=3
In Swap function: x=3           y=2
After Swapping : x=3            y=2
```

Advantages

1. Since arguments are not copied into the new variables, it provides greater time andspace efficiency.
2. The function can change the value of the argument and the change is reflected in the calling function.
3. A function can return only one value. In case we need to return multiple values, we can pass those arguments by reference, so that the modified values are visible in the calling function.

Disadvantage:

1. if inadvertent changes are caused to variables in called function then these changes would bereflected in calling function as original values would have been overwritten.

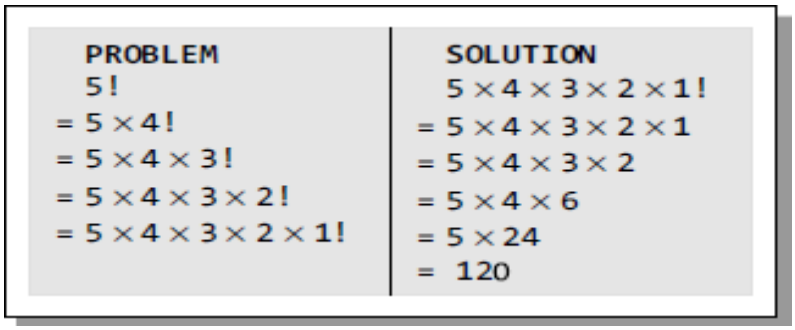
RECURSIVE FUNCTION

Function calls itself again and again is called recursion.

- A recursive function is defined as a function that calls itself to solve a smaller version of its taskuntil a final call is made which does not require a call to itself.

  Every recursive solution has two major cases. They are,

  - Base case - in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - Recursive case - in which first the problem at hand is divided into simpler sub-parts. Second, the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.



Eg:Write a program to calculate the factorial of a given number.

```
#include <stdio.h>
int Fact(int); // FUNCTION DECLARATION
int main()
{
        int num, val;
        printf("\n Enter the number: ");
```

```
        scanf("%d", &num);
        val = Fact(num);
        printf("\n Factorial of %d = %d", num, val);
}
int Fact(int n)
{
        if(n==1)
                return;
                else
                return (n * Fact(n–1));

}
```

Output:

```
Enter the number : 5
Factorial of 5 = 120
```

Types of Recursion

1. Direct Recursion

A function is said to be directly recursive if it explicitly calls itself.

```
int Func (int n)
{
    if (n == 0)
        return n;
    else
        return (Func (n-1));
}
```

2. Indirect Recursion

A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it

```
int Func1 (int n)
{
    if (n == 0)
        return n;
    else
        return Func2(n);
}
int Func2(int x)
{
     return Func1(x-1);
}
```

.

3. Tail Recursion

A recursive function is said to be tail recursive if no operations are pending to be performed when the recursive function returns to its caller.

```
int Func (int n)
{
    if (n == 0)
        return n;
    else
        return (Func (n-1));
}
```

4. Non Tail Recursion:

A recursive function is said to be non tail recursive if operations are pending to be performed when the recursive function returns to its caller.

```
int Fact(int n)
{
if(n==1)
    return 1;else
    return (n * Fact(n–1));
}
```

Advantages

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion works similar to the original formula to solve a problem.
- Recursion follows a divide and conquer technique to solve problems.

disadvantages

- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive program in midstream can be a very slow process.
- Using a recursive function takes more memory and time to execute as compared to itsnon recursive counter part.
- It is difficult to find bugs, particularly while using global variables.

PROGRAM TO DO BINARY SEARCH USING RECURSION

```
#include<stdio.h>
#define size 10
int binsearch(int[], int, int, int);
 int main()
{
        int num, i, key, position;
        int low, high, list[size];
        printf("\nEnter the total number of elements");
        scanf("%d", &num);
        printf("\nEnter the elements of list :");
        for (i = 0; i < num; i++)
        {
                scanf("%d", &list[i]);
        }
        low = 0;
        high = num - 1;
        printf("\nEnter element to be searched : ");
        scanf("%d", &key);
        position = binsearch(list, key, low, high);
        if (position != -1)
        {
                printf("\nNumber present at %d", (position + 1));
        }
        else
                printf("\n The number is not present in the list");
                 return (0);
                                }

         // Binary Search function
        int binsearch(int a[], int x, int low, int high)
        {
                                        int mid;
                                        if (low > high)
                                         return -1;
                                        mid = (low + high) / 2;
                                        if (x == a[mid])

                                          {
                                                return (mid);
                                          }
                                        else if (x < a[mid])

        {
        binsearch(a, x, low, mid - 1);
        }
        else
        {

        binsearch(a, x, mid + 1, high);

        }
```

Output:

Enter the total number of elements : 5
Enter the elements of list : 11 22 33 44 55

Enter element to be searched : 33
Number present at 3

POINTERS

A pointer is a variable that contains the memory location of another variable.

Declaring Pointer Variables

The general syntax of declaring pointer variables can be given as below

data type *ptr name;

Here, data type is the data type of the value that the pointer will point to. For example,

Ex:

```
int x =10
int  *ptr;
ptr=&x
```

ex: program using pointer

```
#include<stdio.h>
int main()
{
int num,*pnum;
pnum=&num;
printf("enter the number");
scanf("%d",&num);
printf("the no that was entered is %d",*pnum);
return 0;
}
```

Output

Enter the number : 10
The number that was entered is : 10

The Pointer Operators:

There are two pointer operators :

1. value at address operator (*)
2. address of operator (&)

Value at address operator (*)

The * is a unary operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

```
q = *m;
```

if m contains the memory address of the variable count, then preceding assignment statement canplaces the value of count into q.

Address of operator (&)

The & is a unary operator that returns the memory address of its operand

```
.m = & count;
```

The preceding assignment statement can be "The memory address of the variable count is places into m".

```
\* Pointer to initialize and print the value and address of variable. *\

# include < stdio.h >
int main( )
{
   int a = 25 ;
   int *b ;
   b = &a ;
   printf("\n Address of a = %u ", & a) ;
   printf("\n Address of a = %u ", b) ;
   printf("\n Address of b = %u ", & b) ;
   printf("\n Value of b = %u ", b) ;
   printf("\n Value of a = %d ", a) ;
   printf("\n Value of a = %d ", *( &a ) ) ;
   printf("\n Value of a = %d ", *b) ;
   return ( 0 );

}
```

Output of the program :
Address of a = 12345
Address of a = 12345
Address of b = 12345
Value of b = 12345
Value of a = 5
Value of a = 5
Value of a = 5

Pointer Arithmetic

There are four arithmetic operators that can be used on pointers: ++, --, +, and –

Valid Pointer Arithmetic Operations

- ✓ Adding a number to pointer.
- ✓ Subtracting a number form a pointer.
- ✓ Incrementing a pointer.
- ✓ Decrementing a pointer.
- ✓ Subtracting two pointers.

Invalid Pointer Arithmetic Operations

- ➢ Addition of two pointers.
- ➢ Division of two pointers.

```
#include <stdio.h>int main()
{
int m = 5, n = 10, q = 0;
int *p1;
int *p2;
int *p3;
p1 = &m;        //printing the address of m
p2 = &n;        //printing the address of n
printf("p1 = %d\n", p1);
printf("p2 = %d\n", p2);
q = *p1+*p2;
printf("*p1+*p2 = %d\n", q);//point 1
p3 = p1-p2;
printf("p1 - p2 = %d\n", p3); //point 2
p1++;
printf("p1++ = %d\n", p1); //point 3
p2--;
printf("p2-- = %d\n", p2); //point 4
//Below line will give ERROR
printf("p1+p2 = %d\n", p1+p2); //point 5return 0;
}
```

OUTPUT: p1 = 2680016
p2 = 2680012
*p1+*p2 = 15
p1-p2 = 1
p1++ = 2680020
p2-- = 2680008

NULL POINTER

null pointer which is a special pointer value and does not point to any value. This means that a nullpointer does not point to any valid memory address.

```
int *ptr = NULL;
```

The null pointer is used in three ways,

1.To stop indirection in a recursive data structure.
2.As an error value
3.As a sentinel value

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

```
  int  *ptr = NULL;
  printf("The value of ptr is %u",ptr);

  return 0;
}
```

Output :

The value of ptr is 0

POINTERS AND ARRAYS :

Syntax**: int *ptr;**

ptr = &arr[0];

Here, ptr is made to point to the first element of the array.

Eg:// program to display an array of given numbers.

```
#include <stdio.h>
int main()
{
int arr[]={1,2,3,4,5,6,7,8,9};
int *ptr1, *ptr2;
ptr1 = arr;
ptr2 = &arr[8];
while(ptr1<=ptr2)
{
printf("%d", *ptr1);
ptr1++;
}
return 0;
}
```

Output
1 2 3 4 5 6 7

ARRAY OF POINTERS:

An array of pointers can be declared as.

datatype *array_name[size];

Eg:int *ptr[10];

The above statement declares an array of 10 pointers where each of the pointer points to aninteger variable.

Example 2://Program on Array of Pointers

```
int main()
{
```

```
int *ptr[10];
int p = 1, q = 2, r = 3, s = 4, t = 5;
ptr[0] = &p;
ptr[1] = &q;
ptr[2] = &r;
ptr[3] = &s;
ptr[4] = &t;
printf("\n %d", *ptr[3]);
return 0;
}
```

OUTPUT:4

Example 2://Program on Array of Pointers

```
int main()
{
int arr1[]={1,2,3,4,5};
int arr2[]={0,2,4,6,8};
int arr3[]={1,3,5,7,9};
int *parr[3] = {arr1, arr2, arr3};
int i;
for(i = 0;i<3;i++)
printf(«%d», *parr[i]);
return 0;
}
```

Output
1 0 1

Applications of Pointers

- Pointers are used to pass information back and forth between functions.
- Pointers enable the programmers to return multiple data items from a function via function arguments.
- Pointers provide an alternate way to access the individual elements of an array.
- Pointers are used to pass arrays and strings as function arguments.
- Pointers are used to create complex data structures, such as trees, linked lists, linked stacks,linkedqueues, and graphs.

PROGRAM TO SORT NAMES

```
#include<stdio.h>
#include<string.h>
int main()
{
int i,j,count;
char str[25][25],temp[25];
puts("How many strings u are going to enter?: ");
```

```
scanf("%d",&count);
puts("Enter Strings one by one: ");
for(i=0;i<=count;i++)
gets(str[i]);
for(i=0;i<=count;i++)
for(j=i+1;j<=count;j++)
strcpy(temp,str[i]);
strcpy(str[i],str[j]);
strcpy(str[j],temp);
}
}
printf("Order of Sorted Strings:");
for(i=0;i<=count;i++)
puts(str[i]);
return 0;
}
```

---

1. Write a program to calculate the GCD of two numbers using recursive functions.#include

```
<stdio.h>
int GCD(int, int);
int main()
{
 int num1, num2, res;
printf("\n Enter the two numbers: ");
scanf("%d %d", &num1, &num2);
res = GCD(num1, num2);
printf("\n GCD of %d and %d = %d", num1, num2, res);
return 0;
}
int GCD(int x, int y)
{
int rem;
rem = x%y;
if(rem==0)
return y;
else
return (GCD(y, rem));
}
```

Output

```
Enter the two numbers : 8 12
GCD of 8 and 12 = 4
```

2. Write a program to print the Fibonacci series using recursion.

```
#include <stdio.h>int
Fibonacci(int);      int
main()
{
int n, i = 0, res;
printf("Enter the number of terms\n");
scanf("%d",&n);
printf("Fibonacci series\n");
for(i = 0; i < n; i++ )
{
res = Fibonacci(i);Stacks 247 printf("%d\t",res);
}
return 0;
}
      int Fibonacci(int n)
                  {
                  if ( n == 0 )
                   return 0;
               else if ( n == 1 )
                     return 1;
                        else
                         return ( Fibonacci(n–1) + Fibonacci(n–2) ))
                  }
```

output:

enter the terms of Fibonacci series 01123

3. Write a program to add two integers using pointers and functions.

```
#include <stdio.h>
void sum (int*, int*, int*);int
main()
{
int num1, num2, total;
printf("\n Enter the first number : ");
scanf("%d", &num1);
printf("\n Enter the second number : ");
scanf("%d", &num2);
sum(&num1, &num2, &total);
printf("\n Total = %d", total);
return 0;
}
void sum (int *a, int *b, int *t)
{
*t = *a + *b;
}
```

Output

```
Enter the first number : 23
Enter the second number : 34
Total = 57
```

UNIT 4 STRUCTURES AND UNION

Structure - Nested structures – Pointer and Structures – Array of structures – Example Program using structures and pointers – Self referential structures – Dynamic memory allocation - Singly linked list – typedef-union storage classes and visibility

STRUCTURES:

- A structure is a user-defined data type that can store related information together. A structure is acollection of variables under a single name.
- the major difference between a structure and an array is that, an array contains related informationof the same data type.
- The variables within a structure are of different data types and each has a name that is used to selectit from the structure.

Features of structures

- Structures can store more than one ***different data type data under a single variable***.
- Structure elements are stored in **successive memory locations**.
- ***Nesting*** of structure is possible.
- Structure elements can be ***passed as argument*** to the function.

Syntax:

//Structure creation

```
struct structurename
{
        Datatype1
        variablename;Datatype2
        variablename;
        .
        .
};
```

//Object Creation

```
struct structname objname;
```

- It is possible to create ***structure pointers***.

Example ://Program to display a point

```
#include<stdio.h>struct point
{
int x,y;
};
void main()
{
struct point p1={2,3};
printf("(%d,%d)",p1.x,p1.y);
}
```

Output:

(2,3)

Initialization of structures:

Syntax:

```
struct struct_name
{
datatype membername1;
datatype membername2;
datatype membername3;
}sturct_var={constant1,constant2,constant3,….};
```

Example:

```
struct student
```

{

int rno;

char name[20];

char course[20];

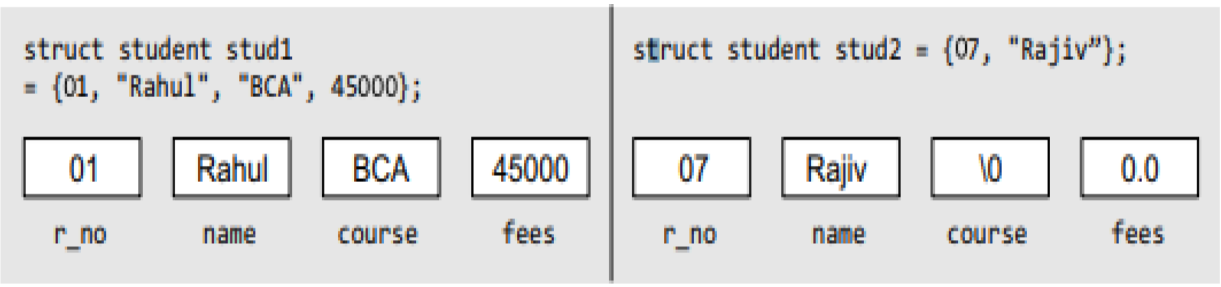
float fees;

}stud1={01,”Rahul”,”BCA”,45000};

or

struct student stud2={02,”Rajiv”};

Fig. illustrates how the values will be assigned to individual fields of the structure



Accessing the members of a structure:

A structure member variable is generally accessed using a ‘.’(dot) operator.

Syntax:

struct_var.membername;

Example:

stud1.rno=01; stud1.name=”Rahul”;stud1.course=”BCA”;stud1.fees=45000;

Receiving user input:

scanf(“%d”,&stud1.rno);

scanf(“%s”,stud1.name);

Displaying output:

printf(“Roll No:%d”,stud1.rno);printf(“Name:%s”,stud1.name);

TYPEDEF DECLARATION:

The ***typedef*** keyword enables the programmer to create a new data type name from an existing data type.

Syntax:

typedef existingdatatype newdatatype;

Example 1:

typedef int INTEGER;INTEGER number=5;

Example 2:

typedef struct student

{

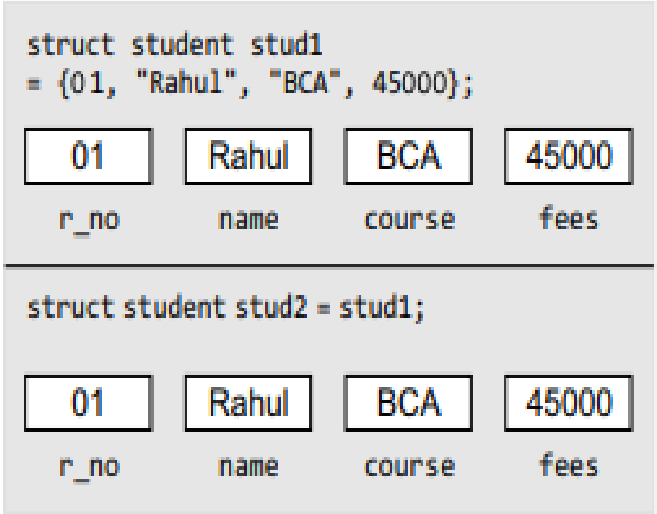
int rno;

char name[20]; char course[20];float fees;

};

student s1; //instead of struct student s1;

COPYING AND COMPARING STRUCTURES:



Values of structure variables

Copy

We can assign a structure to another structure of the same type.struct student
stud1={01,"Rahul","BCA",45000};
struct student stud2=stud1;

Compare:

```
if(stud1.fees==stud2.fees)
        printf("Fees of s2 and s1 are equal");
```

STRUCTURES WITHIN STRUCTURES (NESTED STRUCTURES) :

- Structures can be placed within another structures ie., a structure may contain another structure as itsmember. A structure that contains another structure as its member is called as nested structures.

Example:write a c program to read and display information of students using nested structure

```
#include<stdio.h>
int main()
{
int  day;
int month;
int year;
};struct  student
{
int rollno;
char no[100];
float  fees;
struct  DOB date;
};
struct student stud1;
clrscr();
printf(“enter the roll no”);
scanf(“%d”,&stud1.roll_no);
printf(“enter the name”);
scanf(“%s”,stud1.name);
printf(“enter the fees”);
scanf(“%f “,&stud1.fees);
printf(“enter the  DOB”);
printf(“%d%d%d”,&stud1.date.day,&stud1.date.month,&stud1.date.year);
Printf(“\n ******students details******”);
```

```
Printf("\n ROLL No=%d",stud1.roll_no);
Printf("\n NAME=%s",stud1.name);
Printf("\n FEES = %f",stud1.fees);
Printf("\n DOB =%d-%d-%d",stud1.date.day,stud1.date.month,stud1.date.year);
getch();
return 0;
}
```

Output:

Enter the roll no 01

Enter the name arun

Enter the fees 45000

Enter the DOB 25-09-1991

ARRAYS OF STRUCTURES.

In the above examples, we have seen how to declare a structure and assign values to its datamembers.

The general syntax for declaring an array of structures can be given as,

```
struct struct_name
{
      data_type member_name1;
      data_type member_name2;
      data_type member_name3;
      ......................
};
struct struct_name struct_var[index];
```

Consider the given structure definition.

```
struct student
{
      int r_no;
      char name[20];
      char course[20];
      float fees;
};
```

A student array can be declared by writing,

```
struct student stud[30];
```

Now, to assign values to the `ith` student of the class, we will write

```
stud[i].r_no = 09;
stud[i].name = "RASHI";
```

```
stud[i].course = "MCA";
stud[i].fees = 60000;
```

In order to initialize the array of structure variables at the time of declaration, we can write as follows:

```
struct student stud[3] = {{01, "Aman", "BCA", 45000},{02, "Aryan", "BCA", 60000}, {03,
"John", "BCA", 45000}};
```

4. Write a program to read and display the information of all the students in a class. Then edit the details of the ith student and redisplay the entire information.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
        struct student
        {
                int roll_no;
                char name[80];
                int fees;
                char DOB[80];
        };
        struct student stud[50];
        int n, i, num, new_rolno;
        int new_fees;
        char new_DOB[80], new_name[80];
        clrscr();
        printf("\n Enter the number of students : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n Enter the roll number : ");
                scanf("%d", &stud[i].roll_no);
                printf("\n Enter the name : ");
                gets(stud[i].name);
                printf("\n Enter the fees : ");
                scanf("%d",&stud[i].fees);
                printf("\n Enter the DOB : ");
                gets(stud[i].DOB);
        }
        for(i=0;i<n;i++)
        {
                printf("\n ********DETAILS OF STUDENT %d*******", i+1);
                printf("\n ROLL No. = %d", stud[i].roll_no);
                printf("\n NAME = %s", stud[i].name);
                printf("\n FEES = %d", stud[i].fees);
                printf("\n DOB = %s", stud[i].DOB);
        }
```

```
        return 0;
}
```

Output

```
Enter the number of students : 2
Enter the roll number : 1
Enter the name : kirti
Enter the fees : 5678
Enter the DOB : 9 9 91
Enter the roll number : 2
Enter the name : kangana
Enter the fees : 5678
Enter the DOB : 27 8 91
********DETAILS OF STUDENT 1*******
ROLL No. = 1
NAME = kirti
FEES = 5678
DOB = 9 9 91
********DETAILS OF STUDENT 2*******
ROLL No. = 2
NAME = kangana
FEES = 5678
DOB = 27 8 91
********DETAILS OF STUDENT 1*******
ROLL No. = 1
NAME = kirti
FEES = 5678
DOB = 9 9 91
********DETAILS OF STUDENT 2*******
ROLL No. = 2
NAME = kangana khullar
FEES = 7000
DOB = 27 8 92
```

PASSING STRUCTURES THROUGH POINTERS:

- Passing large structures to functions using the call by value method is very inefficient. Therefore, it is preferred to pass structures through pointers. It is possible to create a pointer to almost any type in C, including the user-defined types.
- It is extremely common to create pointers to structures. A pointer to a structure is a variable that holds the address of a structure. The syntax to declare a pointer to a structure can be given as,

```
struct struct_name
{
data_type member_name1;
data_type member_name2;
data_type member_name3;
.....
}*ptr;

Or

struct struct_name *ptr;
```

- For our student structure, we can declare a pointer variable by writing
- `struct student *ptr_stud, stud;`
-
- The next thing to do is to assign the address of stud to the pointer using the address operator(&), as we would do in case of any other pointer. So to assign the address, we will write
- `ptr_stud = &stud;`
- To access the members of a structure, we can write

`(*ptr_stud).roll_no;`

Write a program to initialize the members of a structure by using a pointer to thestructure.

```
#include<stdio.h>
#include <conio.h>
struct student
{
  int r_no;
  char name[20];
  char course[20];
  int fees;
   };
    int main()
     {
     struct student stud1, *ptr_stud1;
     clrscr();
     ptr_stud1 = &stud1;
     printf("\n Enter the details of the student :
     printf("\n Enter the Roll Number =");
     scanf("%d", &ptr_stud1 -> r_no);
     printf("\n Enter the Name = );
     gets(ptr_stud1 -> name);
     printf("\n Enter the Course = ");
     gets(ptr_stud1 -> course);
```

Output

```
Enter the details of the student:
Enter the RollNumber = 02
Enter the Name = Aditya
Enter the Course = MCA
Enter the Fees = 60000
DETAILS OF THE STUDENT
```

```
 printf("\n Enter the Fees = ");
scanf("%d", &ptr_stud1 -> fees);
printf("\n DETAILS OF THE STUDENT");
printf("\n ROLL NUMBER = %d", ptr_stud1 –> r_no);
printf("\n NAME = %s", ptr_stud1 –> name);
printf("\n COURSE = %s", ptr_stud1 –> course);
printf("\n FEES = %d", ptr_stud1 –> fees);
 return 0;
}
```

SELF-REFERENTIAL STRUCTURES

Self-referential structures are those structures that contain a reference to the data of its same type. That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the sametype as that of the structure. For example, consider the structure node given below.

```
struct node
    {
        int val;
        struct node *next;
    };
```

Here, the structure node will contain two types of data: an integer val and a pointer next. You must be wondering why we need such a structure. Actually, self-referential structure is the foundation of other data structures. We will be using them throughout this book and their purpose will be clearer to you when we discuss linked lists, trees, and graphs.

LINKED LISTS

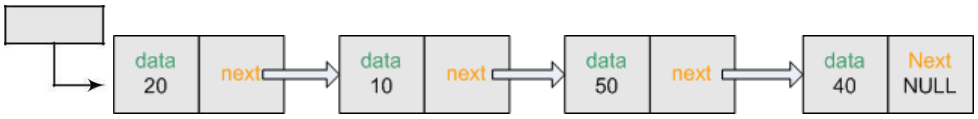
Array is a linear collection of data elements in which the elements are stored in consecutive memorylocations. Its size is fixed.

A linked list does not store its elements in consecutive memory locations and the user can add anynumber of elements to it.

However, unlike an array, a linked list does not allow random access of data. Elements in a

linked listcan be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

A linked list can be perceived as a train or a sequence of nodes in which each node contains one ormore data fields and a pointer to the next node



Start

Simply linked list

Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a self-referential data type.

START - stores the address of the first node in the list
.next - stores the address of its succeeding node.

Declaration of node:

```
struct node
{
int data;
struct node *next;
}
```

Memory Allocation and De-allocation for a Linked List

The Function malloc is most commonly used to attempt to ``grab" a continuous portion of memory. It is defined by:

```
void *malloc(size_t
```

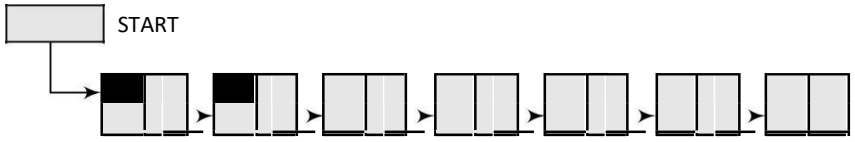
it is usual to use the sizeof() function to specify the number of bytes:

```
Struct node *new_node;
new_node = (struct node*)malloc(sizeof(struct node));
```

SINGLY LINKED Lists

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node,we mean that the node stores the address of the next node in sequence.

A singly linked list allows traversal of data only in one way. Figure 6.7 shows a singly linked list.



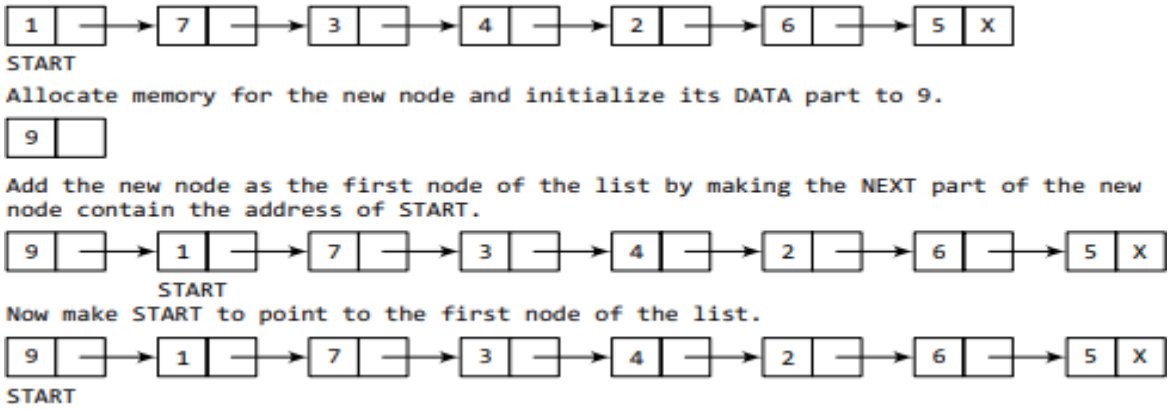
Singly linked list

Inserting a New Node in a Linked List

In this section, we will see how a new node is added into an already existing linked list. We willtake four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node

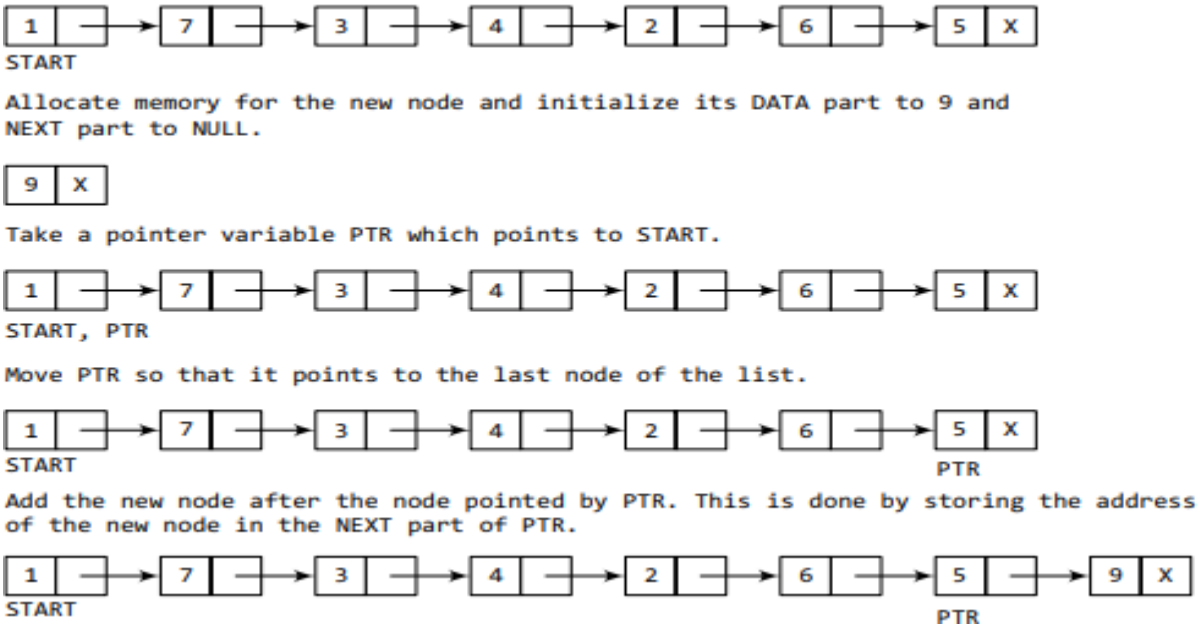
Case 1: Inserting a Node at the Beginning of a Linked List



.

```
struct node *insert_beg(struct node *start)
{
struct node *new_node;int num;
printf("\n Enter the data : ");
scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node));
new_node -> data = num;
new_node -> next = start;
start = new_node;
return start;
}
```

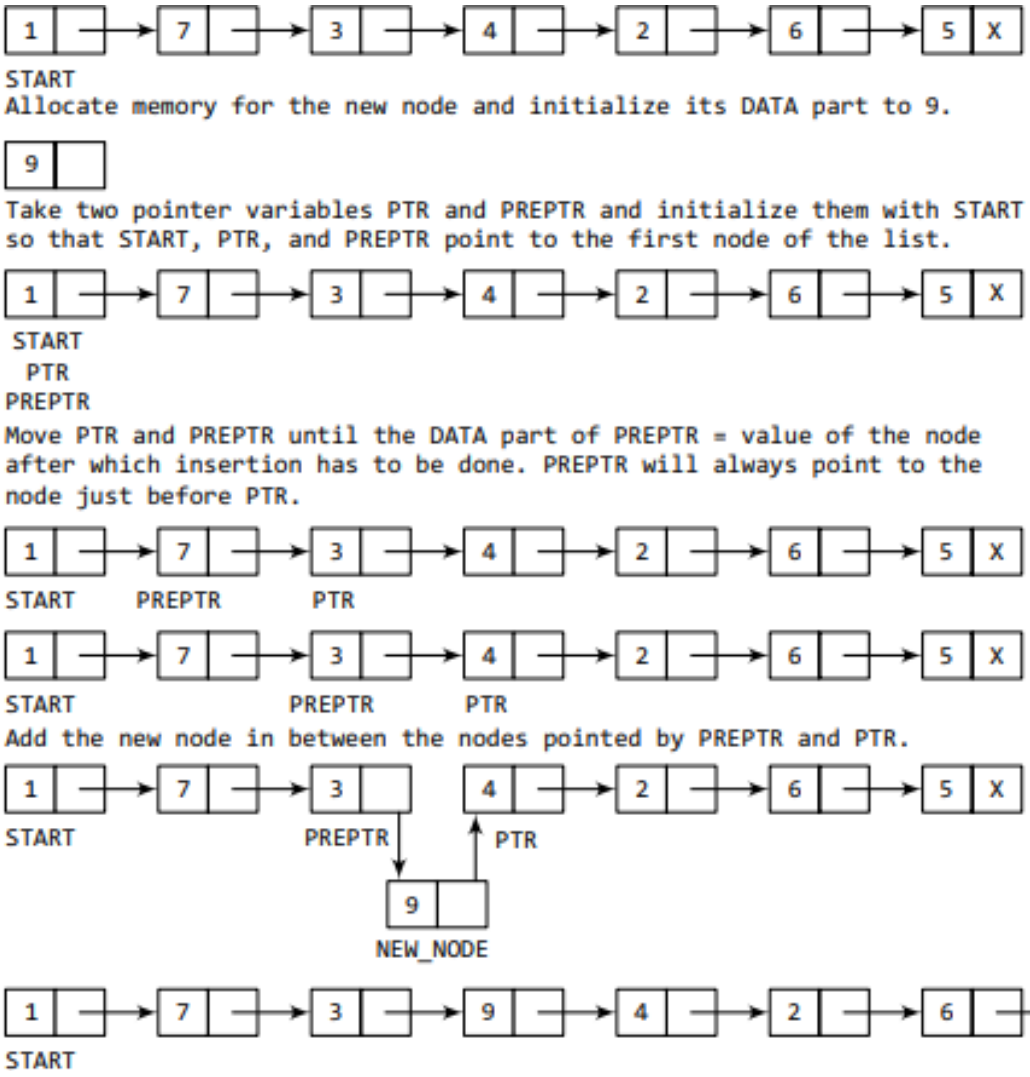
Case 2: Inserting a Node at the End of a Linked List




```
struct node *insert_end(struct node *start)
{
struct node *ptr, *new_node;
```

```
int num;
printf("\n Enter the data : ");
scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node));
new_node -> data = num;
new_node -> next = NULL;
ptr = start;
while(ptr -> next != NULL)ptr = ptr -> next;
ptr -> next = new_node;
return start;
}
```

case 3: Inserting a Node After a Given Node in a Linked List




```
struct node *insert_after(struct node *start)
{
struct node *new_node, *ptr, *preptr;
int num, val;
printf("\n Enter the data : ");
scanf("%d", &num);
printf("\n Enter the value after which the data has to be inserted : ");
scanf("%d", &val);
```

```
new_node = (struct node *)malloc(sizeof(struct node));
new_node -> data = num;
ptr = start;
preptr = ptr;
while(preptr -> data != val)
{
preptr = ptr;
ptr = ptr -> next;
}
preptr -> next=new_node;
new_node -> next = ptr;
return start;
}
```

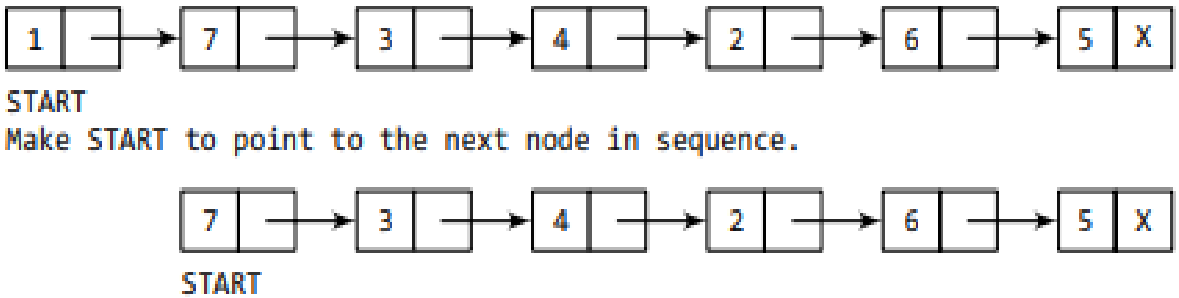
Deleting a Node from a Linked List:

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: The node equal to a given value is deleted.

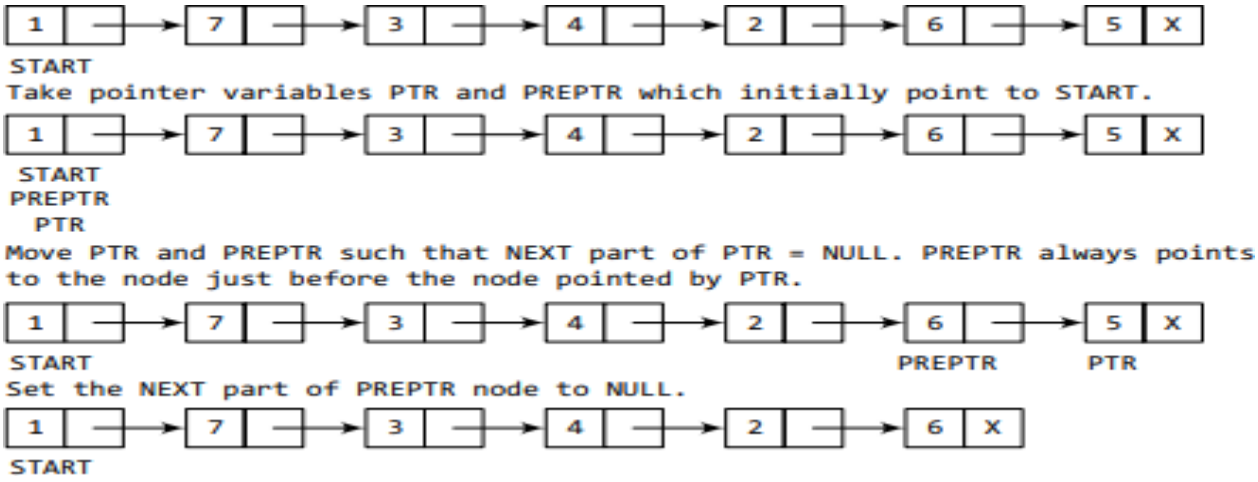
Case 1: Deleting the First Node from a Linked List




```
struct node *delete_beg(struct node *start)
{
```

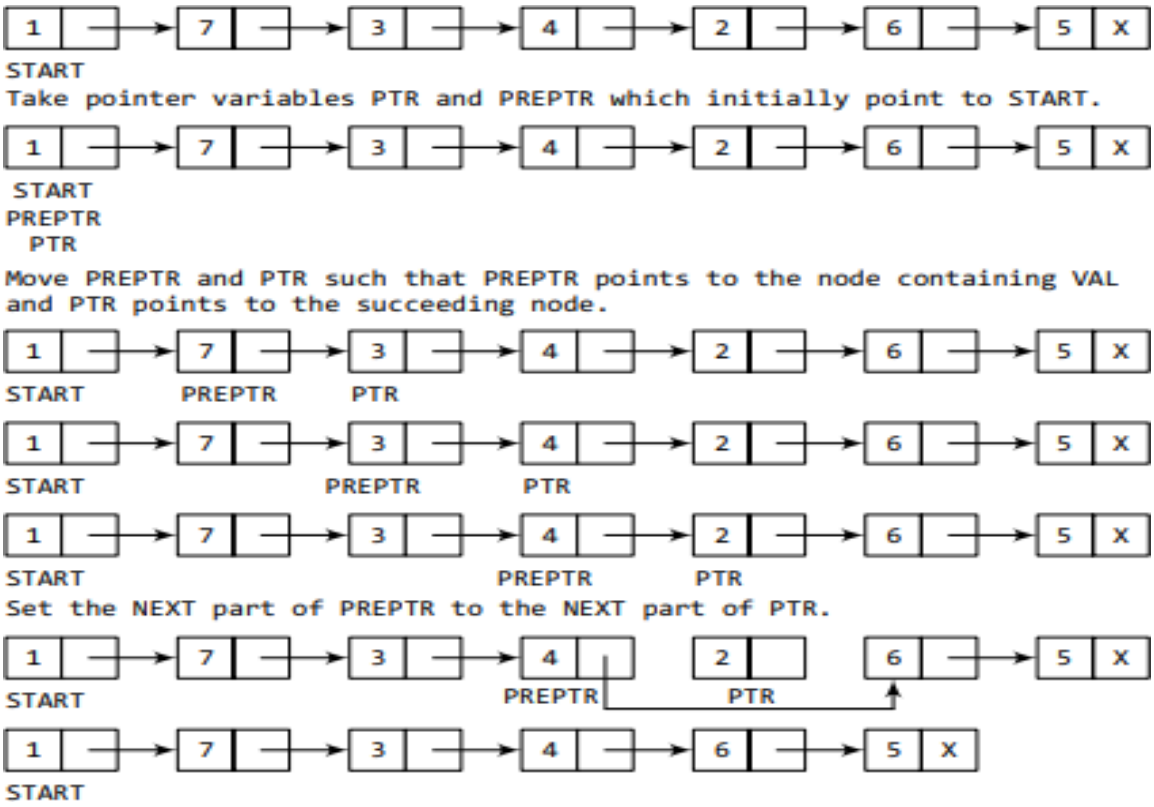
```
struct node *ptr;ptr = start;
start = start -> next;free(ptr);
return start;
}
```

Case 2: Deleting the Last Node from a Linked List




```
struct node *delete_end(struct node *start)
{
struct node *ptr, *preptr;ptr = start;
while(ptr -> next != NULL)
{
preptr = ptr;
ptr = ptr -> next;
}
preptr -> next = NULL;free(ptr);
return start;
}
```

Case 3: Deleting the Node equal to a Given Value in a Linked List




```
struct node *delete_node(struct node *start)
{
        struct node *ptr, *preptr;
        int val;
 printf("\n Enter the value of the node which has to be deleted : ");
        scanf("%d", &val);
          ptr = start;
         if(ptr -> data == val)
    {
```

```
                start = delete_beg(start);
                        return start;
            }
            else
            {

while(ptr -> data != val)
{
preptr = ptr;
ptr = ptr -> next;
}

preptr -> next = ptr -> next;

free(ptr);

return start;

}

}
```

Programming Example

Declare a structure to store information of a particular date.

```
struct date

{

int   day;

int month;

int year;

};
```

Declare a structure to create an inventory record.

```
struct inventory

{

char prod_name[20];

float price;

int stock;
```

```
};
```

Write a program, using an array of pointers to a structure, to read and display the data of students.

```
#include <stdio.h>
 #include <conio.h>
#include <alloc.h>
 struct student
{
int r_no;
char name[20];
char course[20];
int fees;
};
struct student *ptr_stud[10];
int main()
{
int i, n;
    printf("\n Enter the number of students : ");
    scanf("%d", &n);
     for(i=0;i<n;i++)
{
ptr_stud[i] = (struct student *)malloc(sizeof(struct student));
 printf("\nEnter the data for student %d ", i+1);
```

```
printf("\n ROLL NO.: ");
scanf("%d", &ptr_stud[i]–>r_no);
printf("\n NAME: ");
gets(ptr_stud[i]->name);
printf("\ncourse");
gets(ptr->stud[i]->course);
printf("/n FEES");
scanf("%d",&ptr_stud[i]->fees);
}
printf("\n DETAILS OF STUDENT");
for(i=0;i<n;i++)
{
printf(("\n ROLL NO=%d",ptr.stud[i]->r.no);
printf("\n NAME=%s,ptr->stud[i]->name);
printf("\n COURSE=%s,ptr->stud[i]->course);
printf("n FEES=%d",ptr->stud[i]->fees);
}
return 0;
}
```

Output

```
Enter the number ofstudents : 1
Enter the data forstudent 1
ROLL NO.: 01
NAME: Rahul
COURSE:BCA
FEES: 45000
DETAILS OF STUDENT
```

SROLL NO. = 01

NAME = Rahul

COURSE = BCA

FEES = 45000

Write a program that passes a pointer to a structure to a function.

```
#include <stdio.h>
 #include <conio.h>
#include <alloc.h>
struct student
{
int r_no;
char name[20];
char course[20];
int fees;
};
void display (struct student *);
int main()
{
struct student *ptr;
ptr = (struct student *)malloc(sizeof(struct student));
printf("\n Enter the data for the student ");
printf("\n ROLL NO.: ");
scanf("%d", &ptr–>r_no);
printf("\n NAME: ");
gets(ptr–>name);
printf("\n COURSE: ");
gets(ptr–>course);
```

```
}
printf(“/n FEES”);

scanf("%d", &ptr–>fees);
display(ptr);
getch();
 return
}

void display(struct student *ptr)
{
printf("\n DETAILS OF STUDENT");
printf("\n ROLL NO. = %d", ptr–>r_no);
printf("\n NAME = %s", ptr–>name);
printf("\n COURSE = %s ", ptr–>course);
printf("\n FEES = %d", ptr–>fees);
 }
```

Output

```
Enter the data for the student

ROLL NO.: 01
NAME: Rahul
COURSE: BCA
FEES: 45000
DETAILS OF STUDENT
ROLL NO. = 01
NAME = Rahul
COURSE = BCA
FEES = 45000
```