UNIT I BASICS OF C PROGRAMMING

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

1.1 INTRODUCTION TO PROGRAMMING PARADIGMS

Programming paradigms are a way to **classify programming languages** based on their features. Languages can be classified into multiple paradigms.

Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model.

Common programming paradigms include:

- imperative which allows side effects,
- functional which disallows side effects,
- declarative which does not state the order in which operations execute,
- object-oriented which groups code together with the state the code modifies,
- procedural which groups code into functions,
- logic which has a particular style of execution model coupled to a particular style of syntax and grammar, and
- symbolic programming which has a particular style of syntax and grammar.

Machine code

- The lowest-level programming paradigms are **machine code**, which directly represents the instructions (the contents of program memory) as a **sequence of numbers**, and **assembly language** where the machine instructions are represented by **mnemonics** and memory addresses can be given symbolic labels. These are sometimes called first- and second-generation languages.

Procedural languages

The next advance was the development of procedural languages. These third-generation languages (the first described as high-level languages) use vocabulary related to the problem being solved. For example,

- **COmmon Business Oriented Language (COBOL)** – uses terms like file, move and copy.
- **FORmula TRANslation (FORTRAN)** – using mathematical language terminology, it was developed mainly for scientific and engineering problems.
- **ALGOrithmic Language (ALGOL)** – focused on being an appropriate language to define algorithms, while using mathematical language terminology and targeting scientific and engineering problems just like FORTRAN.
- **Programming Language One (PL/I)** – a hybrid commercial-scientific general purpose language supporting pointers.
- **Beginners All purpose Symbolic Instruction Code (BASIC)** – it was developed to enable more people to write programs.
- **C** – a general-purpose programming language, initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.

Features of C Programming Language

- C is a robust language with rich set of built-in functions and operators.
- Programs written in C are efficient and fast.
- C is highly portable, programs once written in C can be run on another machines 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.

Advantages of C

- C is the building block for many other programming languages.
- Programs written in C are highly portable.
- Several standard functions are there (like in-built) that can be used to develop programs.
- C programs are basically collections of C library functions, and it's also easy to add own functions in to the C library.

- The modular structure makes code debugging, maintenance and testing easier.

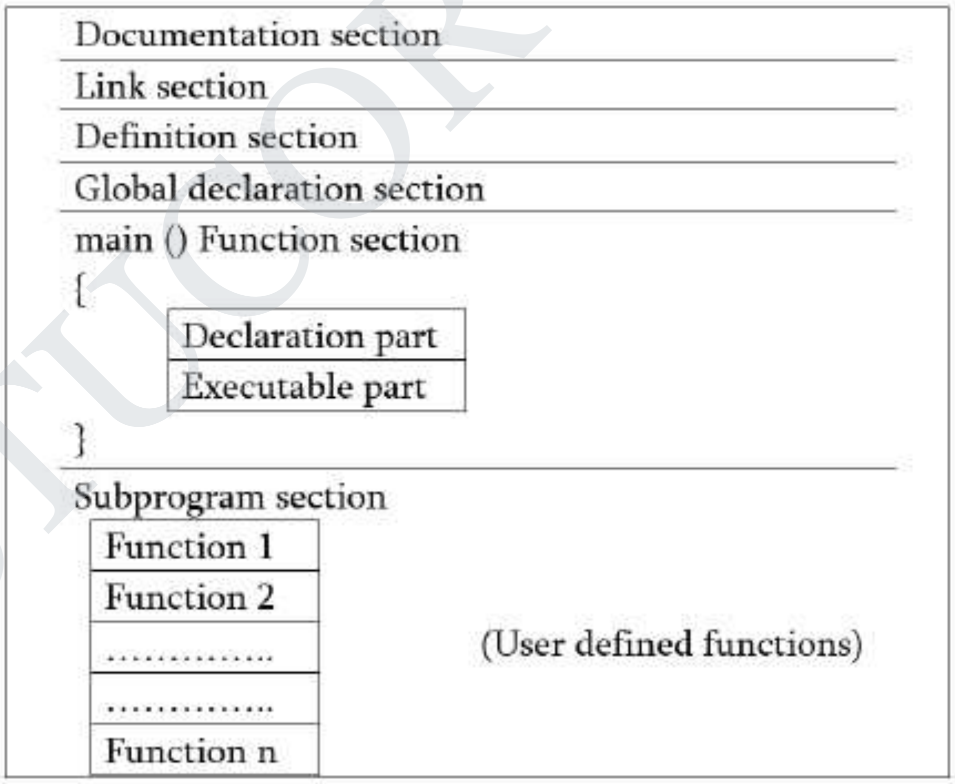
Disadvantages of C

- C does not provide Object Oriented Programming (OOP) concepts.
- There is no concepts of Namespace in C.
- C does not provide binding or wrapping up of data in a single unit.
- C does not provide Constructor and Destructor.

Object-oriented programming

Object-oriented programming (OOP) languages were created, such as **Simula, Smalltalk, C++, C#, Eiffel, PHP, and Java**. In these languages, data and methods to manipulate it are kept as one unit called an object. The only way that another object or user can access the data is via the object's **methods**. Thus, the inner workings of an object may be changed without affecting any code that uses the object.

1.2 STRUCTURE OF C PROGRAM



Documentation section
Link section
Definition section
Global declaration section
main () Function section
{
Declaration part
Executable part
}
Subprogram section
Function 1
Function 2
....
....
Function n
(User defined functions)

1. **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.

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

3. **Definition section:** The definition section defines all symbolic constants such using the **#define directive.**
4. **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.**
5. **main () function section:** Every C program must have one main function section. This section contains two parts; declaration part and executable part
   1. **Declaration part:** The declaration part declares all the **variables** used in the executable part.
   2. **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.
6. **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.

1.3 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 type for 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 specifying the 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 Names

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:

Data Types	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.

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	

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
}
```

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;
}
```

1.4 STORAGE CLASSES

Storage classes are used to **define scope and life time of a variable**. There are four storage classes in C programming.

- auto
- extern
- static
- register

Storage Classes	Storage Place	Default Value	Scope	Life-time
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of main program, May be declared anywhere in the program
static	RAM	Zero	Local	Till the end of main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within function

1) auto

The **auto keyword** is applied to **all local variables automatically**. It is the **default storage class** that is why it is known as automatic variable.

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

```
int a=10;
auto int b=10;//same like above
printf("%d %d",a,b);
return 0;
}
```

Output:

```
10 10
```

2) register

The register variable **allocates memory in register than RAM**. Its size is **same of register size**. It has a **faster access** than other variables.

It is **recommended** to use register variable only for **quick access** such as in counter.

We can't get the address of register variable.

Example: register int counter=0;

3) static

The **static** variable is **initialized only once and exists** till the end of the program. It **retains its value between multiple functions call.**

The static variable has the **default value 0** which is provided by compiler.

Example:

```
#include<stdio.h>
int  func()
{
  static int i=0;//static variable
  int j=0;//local variable
  i++;
  j++;
  printf("i= %d and j= %d\n", i, j);
}
int main() {
 func();
 func();
 func();
return 0;
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

4) extern

The extern variable is **visible to all the programs**. It is used if **two or more files are sharing same variable or function**.

Example: extern int counter=0;

1.5 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

1) 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 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
```

2) 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	\v	vertical tab
\b	backspace	\’	single quote
\f	form feed	\”	double quote
\n	new line	\\	backslash
\r	carriage return	\0	null
\t	horizontal tab		

1.6 ENUMERATION CONSTANTS

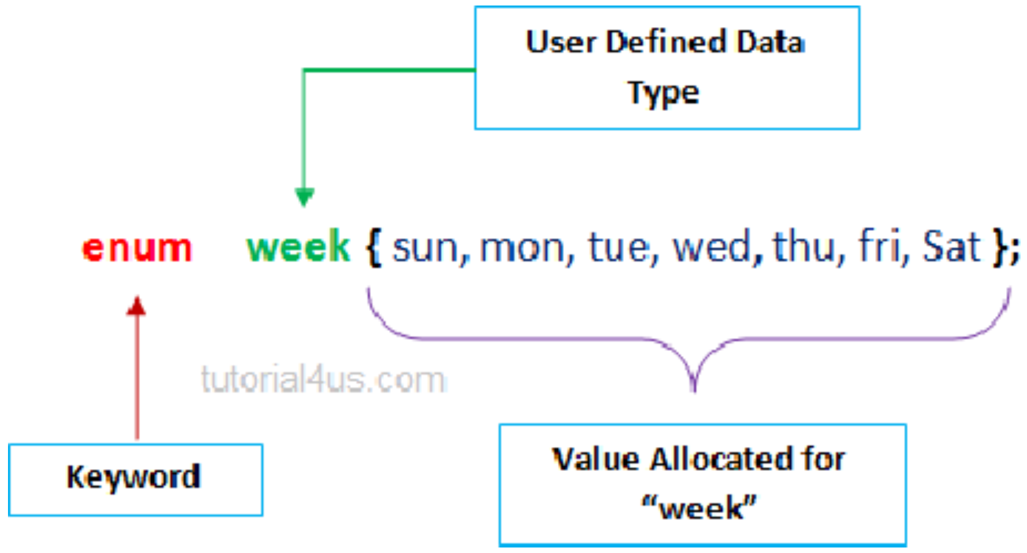
An **enum** is a keyword, it is an user defined data type. All properties of integer are applied on Enumeration data type so **size of the enumerator data type is 2 byte**. It **work like the Integer.**

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.


User Defined Data Type
enum week { sun, mon, tue, wed, thu, fri, Sat };
tutorial4us.com
Keyword
Value Allocated for "week"

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 of Enumeration in C:

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
```

1.7 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.

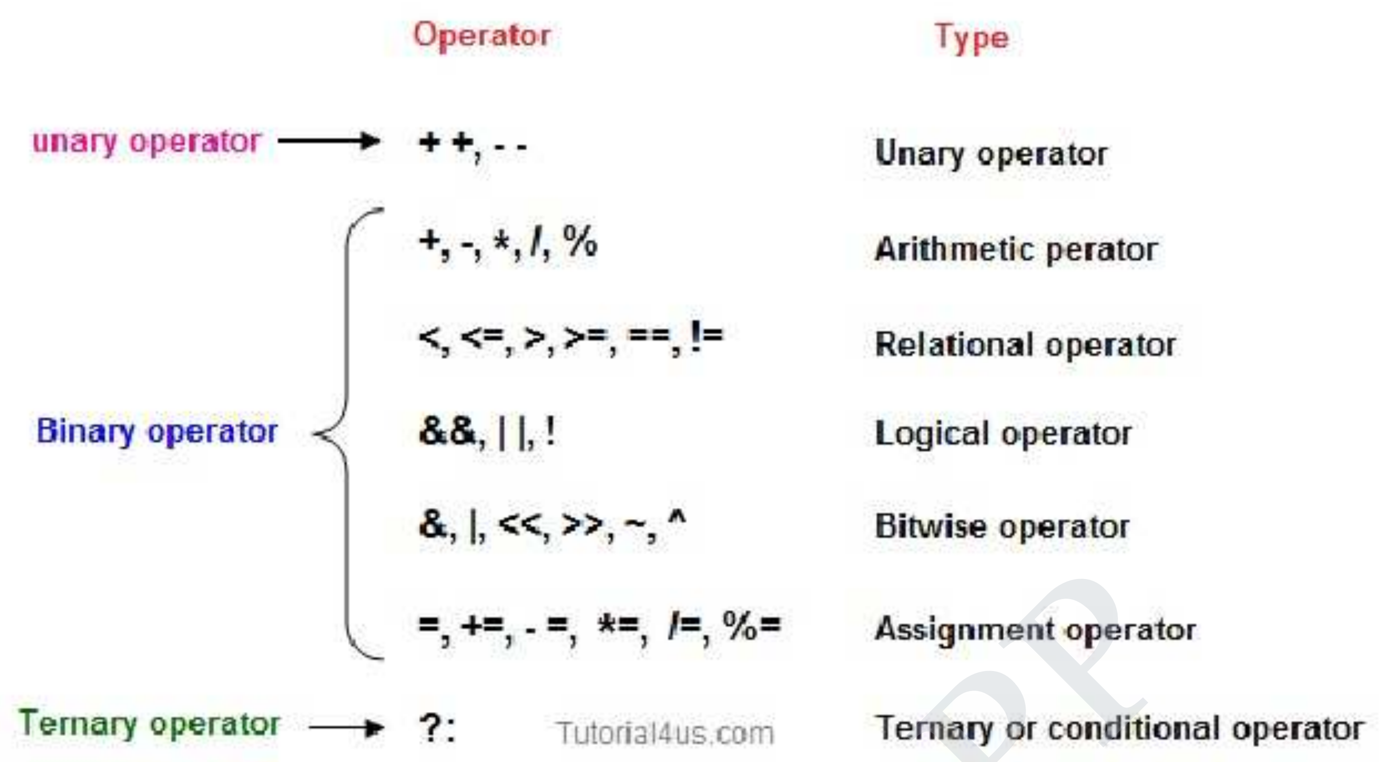
A list of 32 keywords in c language is given below:

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

1.8 OPERATORS: PRECEDENCE AND ASSOCIATIVITY

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



	Operator	Type
unary operator →	+ +, - -	Unary operator
Binary operator	+, -, *, /, %	Arithmetic perator
	<, <=, >, >=, ==, !=	Relational operator
	&&, | |, !	Logical operator
	&, |, <<, >>, ~, ^	Bitwise operator
	=, +=, - =, *=, /=, %=	Assignment operator
Ternary operator →	?:	Ternary or conditional operator

Tutorial4us.com

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 **A** hold 8 and **B** hold 3.

Operators	Example (int A=8, B=3)	Result
<	A<B	False
<=	A<=10	True
>	A>B	True
>=	A<=B	False
==	A== B	False

!=	A!=(-4)	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

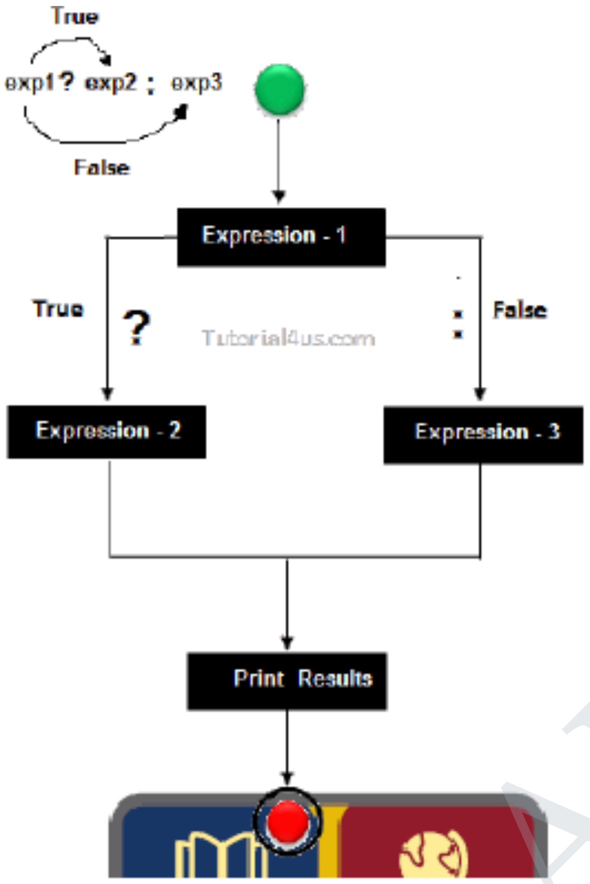
Advantage of Ternary Operator

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.

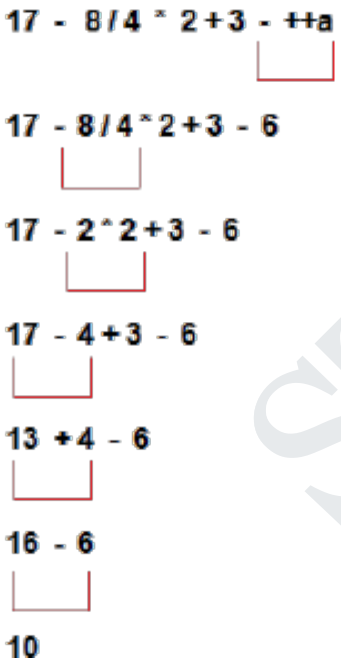
Associativity

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

Precedence	Operator	Operator Meaning	Associativity
1	() [] -> .	function call array reference structure member access structure member access	Left to Right
2	! ~ + - ++ -- & * sizeof (type)	negation 1's complement Unary plus Unary minus incre ment operator decrement operator address of operator pointer returns size of a variable type conversion	Right to Left
3	* / %	multiplication division remainder	Left to Right
4	+ -	addition subtraction	Left to Right
5	<< >>	left shift right shift	Left to Right
6	< <= > >=	less than less than or equal to greater than greater than or equal to	Left to Right

7	== !=	equal to not equal to	Left to Right
8	&	bitwise AND	Left to Right
9	^	bitwise EXCLUSIVE OR	Left to Right
10	|	bitwise OR	Left to Right
11	&&	logical AND	Left to Right
12	||	logical OR	Left to Right
13	?:	conditional operator	Left to Right
14	= *= /= %= += -= &= ^= |= <<= >>=	assignment assign multiplication assign division assign remainder assign additon assign subtraction assign bitwise AND assign bitwise XOR assign bitwise OR assign left shift assign right shift	Right to Left
15	,	separator	Left to Right

Example:



```
17 - 8/4 * 2+3 - ++a
17 - 8/4*2+3 - 6
17 - 2*2+3 - 6
17 - 4+3 - 6
13 +4 - 6
16 - 6
10
```

1.9 INPUT/OUTPUT STATEMENTS

Majority of the programs take data as input, and then after processing the processed data is being displayed which is called information. In C programming you can use **scanf() and printf()** predefined function to read and print data.

Managing Input/Output

I/O operations are useful for a program to interact with users. **stdlib** is the standard C library for input-output operations. While dealing with input-output operations in C, there are two important streams that play their role. These are:

- Standard Input (stdin)
- Standard Output (stdout)

Standard input or **stdin** is used for **taking input from devices such as the keyboard** as a data stream. Standard output or **stdout** is used for **giving output to a device** such as a monitor. For using **I/O functionality**, programmers must include **stdio header-file** within the program.

Reading Character In C

The easiest and simplest of all I/O operations are **taking a character as input** by reading that character from standard input (keyboard). **getchar()** function can be used to **read a single character**. This function is **alternate to scanf()** function.

Syntax:

```
var_name = getchar();
```

Example:

```
#include<stdio.h>
void main()
{
        char title;
        title = getchar();
}
```

There is another function to do that task for files: **getc which is used to accept a character from standard input.**

Syntax:

```
int getc(FILE *stream);
```

Writing Character In C

Similar to getchar() there is another function which is used to write characters, but one at a time.

Syntax:

```
putchar(var_name);
```

Example:

```
#include<stdio.h>
void main()
{
        char result = 'P';
        putchar(result);
        putchar('\n');
}
```

Similarly, there is another function **putc** which is used for **sending a single character to the standard output.**

Syntax:

```
int putc(int c, FILE *stream);
```

Formatted Input

It refers to an input data which has been **arranged in a specific format**. This is possible in C using scanf(). We have already encountered this and familiar with this function.

Syntax:

```
scanf("control string", arg1, arg2, ..., argn);
```

Format specifier:

Format specifier	Type of value
%d	Integer
%f	Float
%lf	Double
%c	Single character
%s	String
%u	Unsigned int
%ld	Long int
%lf	Long double

Example:

```
#include<stdio.h>
void main()
{
        int var1= 60;
        int var1= 1234;
        scanf("%2d %5d", &var1, &var2);
}
```

Input data items should have to be separated by spaces, tabs or new-line and the punctuation marks are not counted as separators.

Reading and Writing Strings in C

There are two popular library functions **gets() and puts()** provides to deal with strings in C.

gets: The char *gets(char *str) **reads a line from stdin** and keeps the string pointed to by the str and is terminated when the new line is read or EOF is reached. The declaration of gets() function is:

Syntax:

```
char *gets(char *str);
```

where str is a pointer to an array of characters where C strings are stored.

puts: The function – int puts(const char *str) is used to **write a string to stdout** but it does not include null characters. A new line character needs to be appended to the output. The declaration is:

Syntax:

```
int puts(const char *str);
```

where str is the string to be written in C.

1.10 ASSIGNMENT STATEMENTS

The assignment statement has the following form:

```
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*/
```

1.11 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.

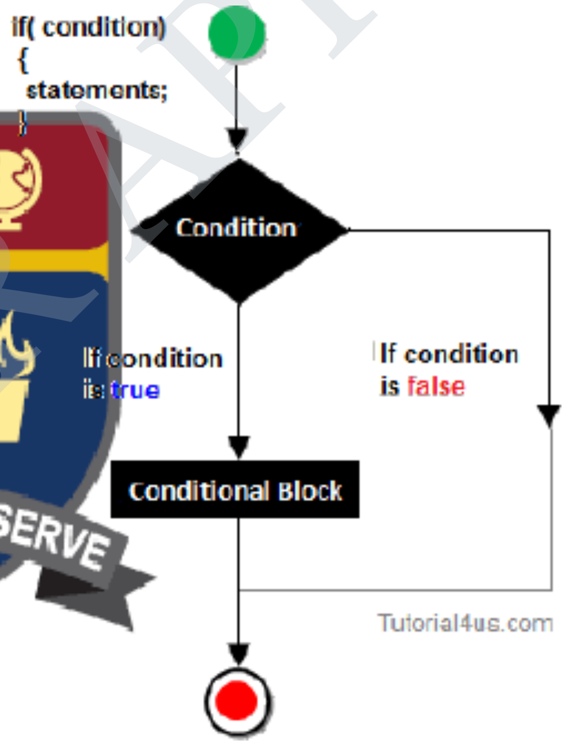
- if
- if-else
- switch

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
}
```



- 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>

void main()

{

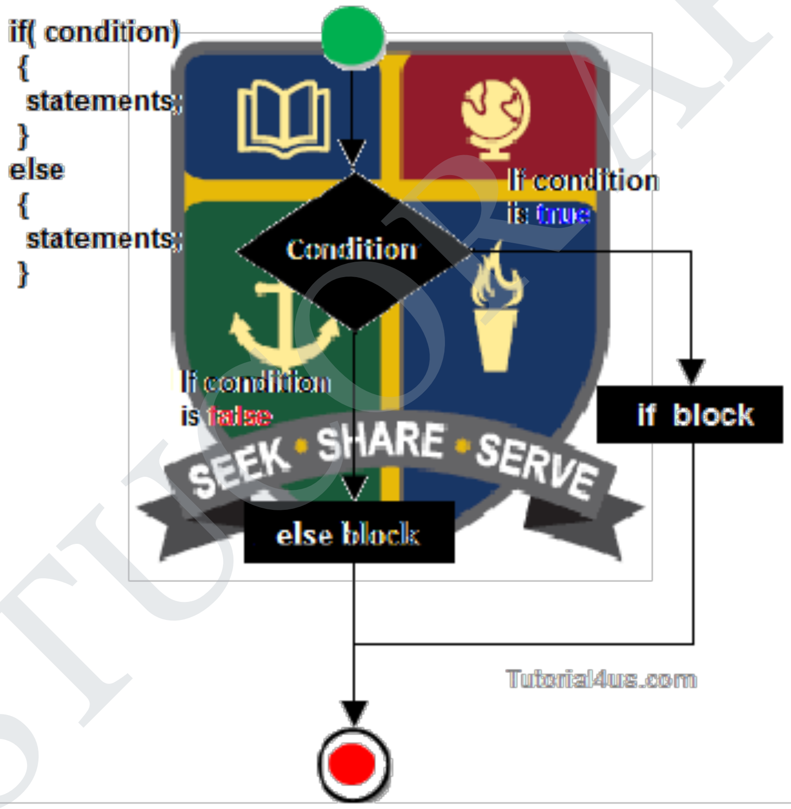
```
int time=10;
if(time>12)
{
printf("Good morning")
}
}
```

Output:

```
Good morning
```

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.



```
if( condition)
 {
  statements;
 }
else
 {
  statements;
 }
```

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 time=10;
if(time>12)
{
printf("Good morning")
}
else
{
printf("good after noon")
}
}
```

Output:

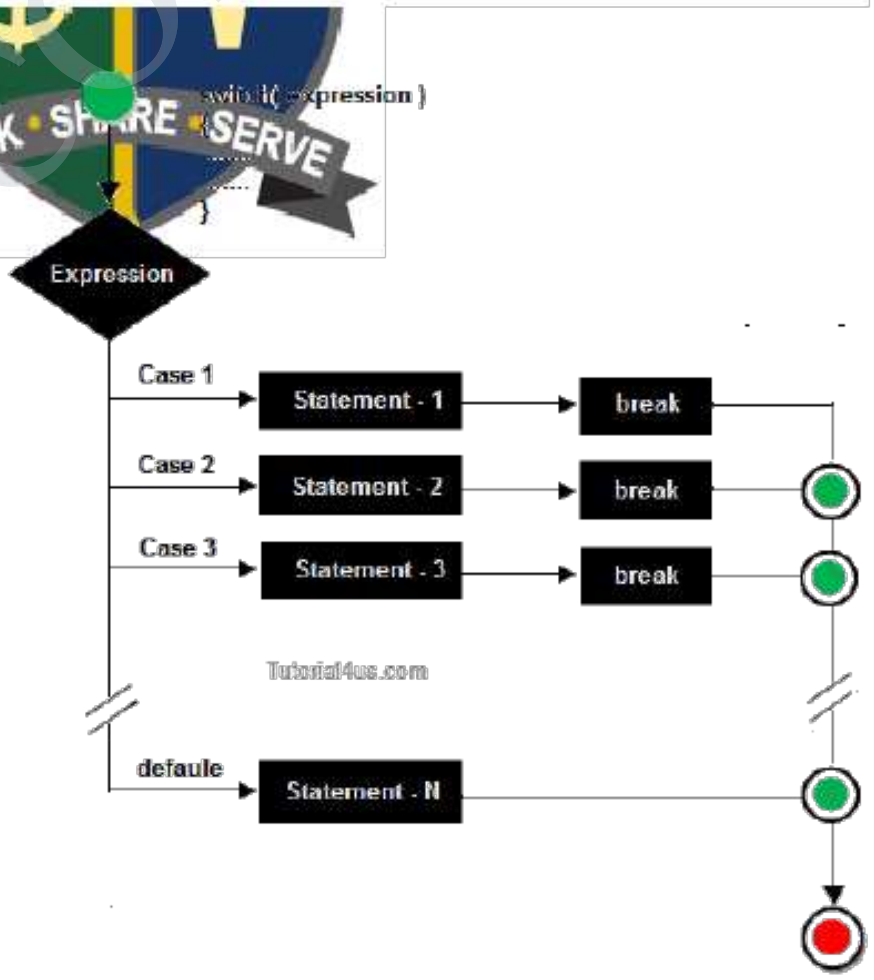
```
Good morning
```

1.12 SWITCH STATEMENT

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 .

Example:

```
#include<stdio.h>
void main()
{
int a;
printf("Please enter a no between 1 and 5: ");
scanf("%d",&a);
switch(a)
{
case 1:
printf("You chose One");
break;
case 2:
printf("You chose Two");
break;
case 3:
printf("You chose Three");
break;
case 4:
printf("You chose Four");
break;
case 5:
printf("You chose Five.");
break;
default :
printf("Invalid Choice. Enter a no between 1 and 5");
break;
}
}
```

Output:

```
Please enter a no between 1 and 5  3
You choice three
```

1.13 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:

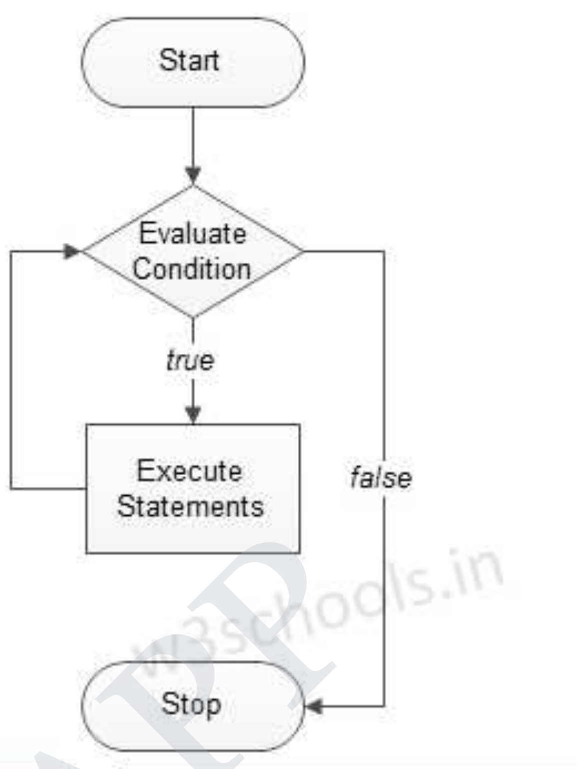
- while loops
- do while loops
- for loops

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;
}
```



Example:

```
#include<stdio.h>
 int main ()
{
   /* local variable Initialization */
   int n = 1,times=5;
   /* while loops execution */
   while( n <= times )
   {
      printf("C while loops: %d\n", n);
      n++;
   }
   return 0;
}
```

Output:

```
C while loops:1
C while loops:2
C while loops:3
C while loops:4
C while loops:5
```

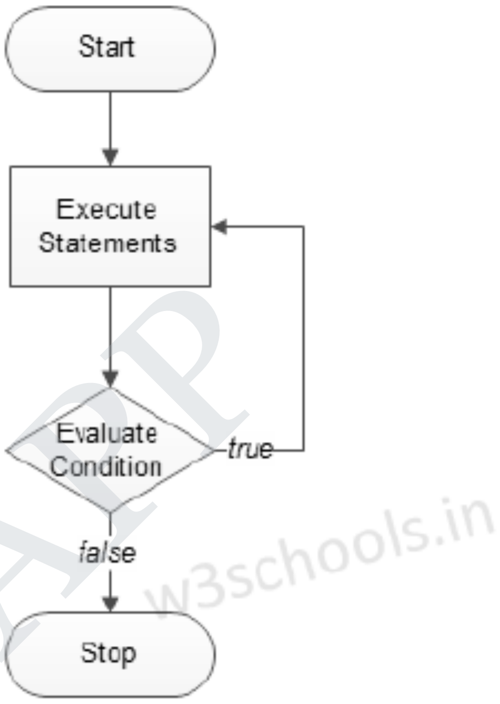
Do..while loops:

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 );
```



Example:

```
#include<stdio.h>
 int main ()
{
   /* local variable Initialization */
   int n = 1,times=5;
   /* do loops execution */
   do
   {
      printf("C do while loops: %d\n", n);
      n = n + 1;
   }while( n <= times );
   return 0;
}
```

Output:

```
C do while loops:1
C do while loops:2
C do while loops:3
```

C do while loops:4

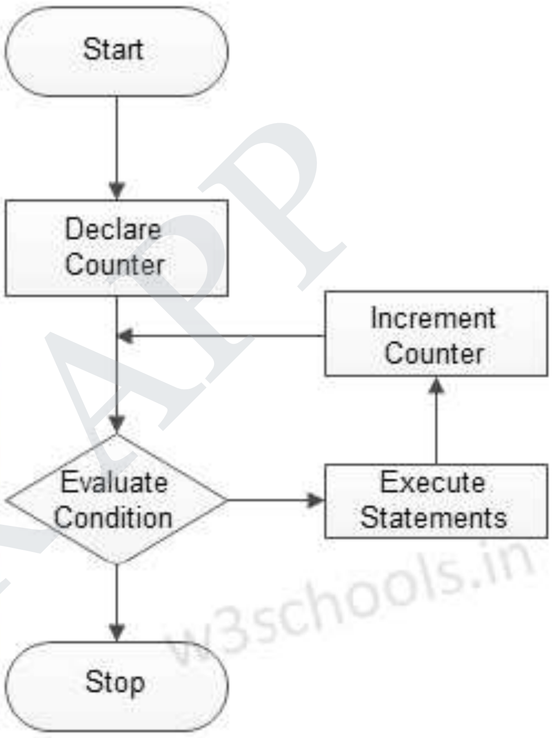
C do while loops:5

for loops

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);
}
```



Example:

```
#include<stdio.h>
 int main ()
{
  /* local variable Initialization */
  int n,times=5;;

   /* for loops execution */
   for( n = 1; n <= times; n = n + 1 )
   {
      printf("C for loops: %d\n", n);
   }
    return 0;
}
```

Output:

```
C for loops:1
C for loops:2
C for loops:3
C for loops:4
C for loops:5
```

C Loop Control Statements

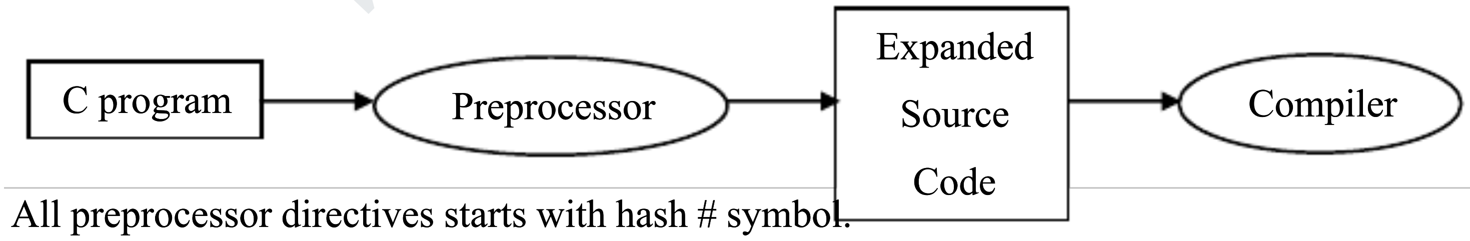
Loop control statements are used to change the normal sequence of execution of the loop.

Statement	Syntax	Description
break statement	break;	It is used to **terminate loop** or **switch statements**.
continue statement	continue;	It is used to **suspend the execution of current loop iteration** and **transfer control** to the loop for the next iteration.

goto statement	goto labelName;
	labelName: statement;
		It **transfers current program** execution sequence to some other part of the program.

1.14 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 directives	Purpose	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 micro substitution. It can use any basic data type.	#define PI 3.14
3	#undef	Used to undefine the constant or macro defined by #define.	#define PI 3.14 #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 not 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 c ompile
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

1.15 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.

- Then 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 better code.

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 and included 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. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line 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 – Example Program**:** Computing Mean, Median and Mode - Two dimensional arrays – Example Program: Matrix Operations (Addition, Scaling, Determinant and Transpose) - String operations: length, compare, concatenate, copy – Selection sort, linear and binary search

INTRODUCTION TO ARRAYS: DECLARATION, INITIALIZATION – ONE DIMENSIONAL ARRAY

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.

```
data_type array_name[array_size];
```

Now, let us see the example to declare array.

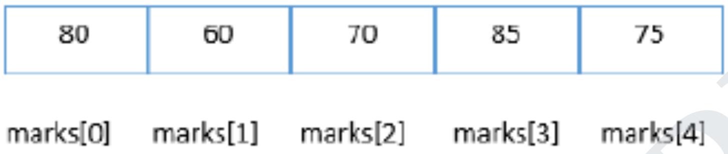
int marks[5];

Here, int is the data_type, marks is the array_name and 5 is the array_size.

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;
```



Initialization of Array

Example 1:

```
#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.

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

Example 2:

```
#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.

```
data_type array_name[size1][size2];
```

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

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.

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

Example:

```
#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 OPERATIONS

What is meant by String?

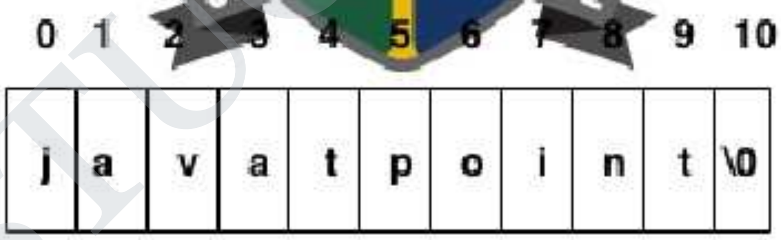
String in C language is an **array of characters** that is **terminated by \0** (null character). There are two ways to declare string in c language.

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.

Example:

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.

Example:

```
#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.

Example:

```
#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**.

Example:

```
#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
```

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

3.1 INTRODUCTION TO FUNCTIONS

C function is a **self-contained block of statements** that can be executed **repeatedly** whenever we need it.

Benefits of using function in C

- The function provides modularity.
- The function provides reusable code.
- In large programs, debugging and editing tasks is easy with the use of functions.
- The program can be modularized into smaller parts.
- Separate function independently can be developed according to the needs.

There are two types of functions in C

- **Built-in(Library) Functions**

These functions are provided by the system and stored in the library, therefore it is also called Library Functions.

e.g. scanf(), printf(), strcpy, strlwr, strcmp, strlen, strcat etc.

To use these functions, you just need to include the appropriate C header files.

- **User Defined Functions**

These functions are defined by the user at the time of writing the program.

Parts of Function

1. Function Prototype (function declaration)
2. Function Definition
3. Function Call

1. Function Prototype

Syntax:

```
datatype functionname(parameter list)
```

Example:

```
int addition();
```

2. Function Definition

Syntax:

```
returnType functionName(Function
arguments)
{
  //body of the function
}
```

Example:

```
int addition()
{

}
```

3. Calling a function in C

Syntax:

```
functionName(Function arguments)
```

Program to illustrate Addition of Two Numbers using User Defined Function

Example:

```
#include<stdio.h>
/* function declaration */
int addition();
int main()
{
   /* local variable definition */
   int answer;
```

```
     /* calling a function to get addition value */
   answer = addition();
   printf("The addition of two numbers is: %d\n",answer);
   return 0;
}
/* function returning the addition of two numbers */
int addition()
{
   /* local variable definition */
   int num1 = 10, num2 = 5;
   return num1+num2;
}
```

Output:

```
The addition of two numbers is: 15
```

3.2 PARAMETER PASSING: PASS BY VALUE, PASS BY REFERENCE

When a function gets executed in the program, the execution control is transferred from calling function to called function and executes function definition, and finally comes back to the calling function. When the execution control is transferred from calling function to called function it may carry one or more number of data values. These data values are called as **parameters**.

Parameters are the data values that are passed from calling function to called function.

In C, there are two types of parameters and they are as follows...

- Actual Parameters
- Formal parameters

The **actual parameters** are the parameters that are specified in calling function.

The **formal parameters** are the parameters that are declared at called function. When a function gets executed, the copy of actual parameter values are copied into formal parameters.

In C Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...

- Call by value

- Call by reference

Call by Value

In **call by value** parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. **The changes made on the formal parameters does not affect the values of actual parameters** . That means, after the execution control comes back to the calling function, the actual parameter values remains same. For example consider the following program...

Example:

```
#include <stdio.h>
#include<conio.h>
void main(){
  int num1, num2 ;
  void swap(int,int) ; // function declaration
  clrscr() ;
  num1 = 10 ;
  num2 = 20 ;
    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
    swap(num1, num2) ; // calling function
    printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
  getch() ;
}
void swap(int a, int b)  // called function
{
  int temp ;
  temp = a ;
  a = b ;
  b = temp ;
}
```

Output:

```
Before swap: num1 = 10, num2 = 20
After swap: num1 = 10, num2 = 20
```

swap: num1 = 10, num2 = 20

In the above example program, the variables **num1** and **num2** are called actual parameters and the variables **a** and **b** are called formal parameters. The value of **num1** is copied into **a** and the value of num2 is copied into **b**. The changes made on variables **a** and **b** does not affect the values of **num1** and **num2**.

Call by Reference

In **Call by Reference** parameter passing method, the memory location address of the actual parameters is copied to formal parameters. This address is used to access the memory locations of the actual parameters in called function. In this method of parameter passing, the formal parameters must be **pointer** variables.

That means in call by reference parameter passing method, the address of the actual parameters is passed to the called function and is received by the formal parameters (pointers). Whenever we use these formal parameters in called function, they directly access the memory locations of actual parameters. So **the changes made on the formal parameters effects the values of actual parameters**. For example consider the following program...

Example:

```
#include <stdio.h>
#include<conio.h>
void main(){
  int num1, num2 ;
  void swap(int *,int *) ; // function declaration
  clrscr() ;
  num1 = 10 ;
  num2 = 20 ;
    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
  swap(&num1, &num2) ; // calling function
    printf("\nAfter swap: num1 = %d, num2 = %d", num1, num2);
  getch() ;
}
void swap(int *a, int *b)  // called function
{
  int temp ;
  temp = *a ;
  *a = *b ;
  *b = temp ;
}
```

Output:

```
Before swap: num1 = 10, num2 = 20
After swap: num1 = 20, num2 = 10
```

In the above example program, the addresses of variables **num1** and **num2** are copied to pointer variables **a** and **b**. The changes made on the pointer variables **a** and **b** in called function effects the values of actual parameters **num1** and **num2** in calling function.

3.3 BUILT-IN FUNCTIONS (STRING FUNCTIONS, MATH FUNCTIONS)

String Functions

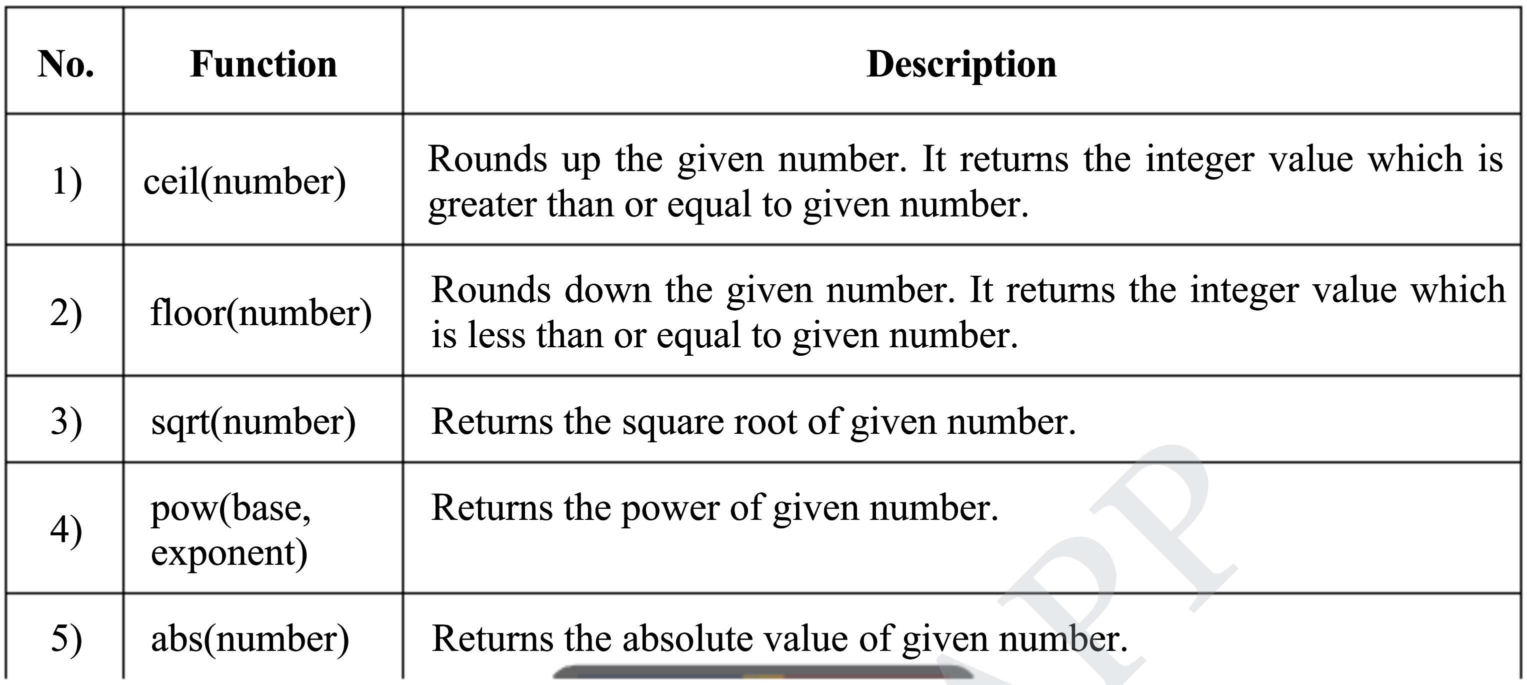
There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	Returns the length of string name.
2)	strcpy(destination, source)	Copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	Concatenates or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	Compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	Returns reverse string.
6)	strlwr(string)	Returns string characters in lowercase.
7)	strupr(string)	Returns string characters in uppercase.

Math Functions

C Programming allows us to perform mathematical operations through the functions defined in **<math.h>** header file. The <math.h> header file contains various methods for performing mathematical operations such as sqrt(), pow(), ceil(), floor() etc.

There are various methods in math.h header file. The commonly used functions of math.h header file are given below.



No.	Function	Description
1)	ceil(number)	Rounds up the given number. It returns the integer value which is greater than or equal to given number.
2)	floor(number)	Rounds down the given number. It returns the integer value which is less than or equal to given number.
3)	sqrt(number)	Returns the square root of given number.
4)	pow(base, exponent)	Returns the power of given number.
5)	abs(number)	Returns the absolute value of given number.

Example:

```
#include<stdio.h>
#include <math.h>
int main(){
printf("\n%f",ceil(3.6));
printf("\n%f",ceil(3.3));
printf("\n%f",floor(3.6));
printf("\n%f",floor(3.2));
printf("\n%f",sqrt(16));
printf("\n%f",sqrt(7));
printf("\n%f",pow(2,4));
printf("\n%f",pow(3,3));
printf("\n%d",abs(-12));
 return 0;
}
```

Output:

```
4.000000
4.000000
3.000000
```

```
3.000000
4.000000
2.645751
16.000000
27.000000
12
```

3.4 RECURSION

When function is called within the same function, it is known as **recursion** in C. The function which calls the same function, is known as **recursive function**.

A function that calls itself, and doesn't perform any task after function call, is know as **tail recursion**. In tail recursion, we generally call the same function with return statement. An example of tail recursion is given below.

Let's see a simple example of recursion.

```
recursionfunction(){
recursionfunction();//calling self function
}
```

Example:

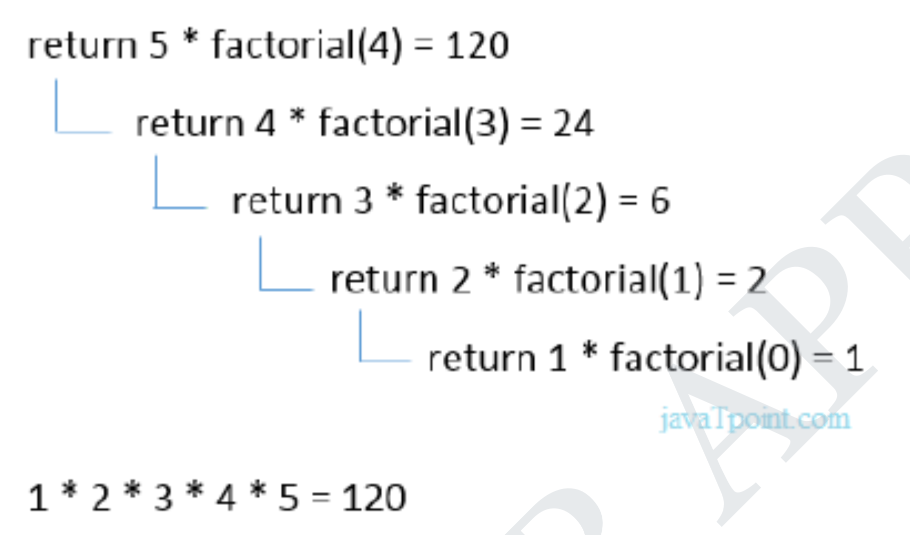
```
#include<stdio.h>
int factorial (int n)
{
   if ( n < 0)
      return -1; /*Wrong value*/
   if (n == 0)
      return 1; /*Terminating condition*/
   return (n * factorial (n -1));
}
int main(){
int fact=0;
fact=factorial(5);
printf("\n factorial of 5 is %d",fact);
```

```
    return 0;
    }
```

Output:

```
factorial of 5 is 120
```

We can understand the above program of recursive method call by the figure given below:



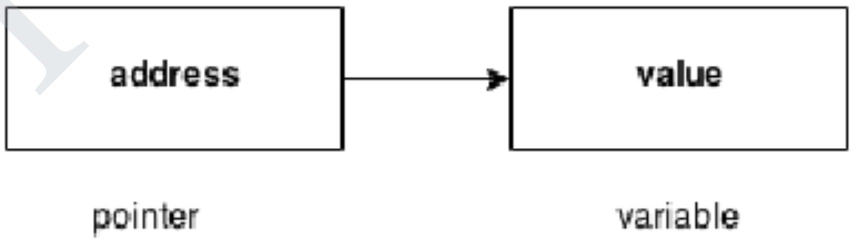
```
return 5 * factorial(4) = 120
  └── return 4 * factorial(3) = 24
        └── return 3 * factorial(2) = 6
              └── return 2 * factorial(1) = 2
                    └── return 1 * factorial(0) = 1

1 * 2 * 3 * 4 * 5 = 120
```

Fig: Recursion

3.5 POINTERS

The **pointer in C language** is a variable, it is also known as locator or indicator that points to an address of a value.



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can **return multiple values from function** using pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many usage of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	address of operator	determines the address of a variable.
* (asterisk sign)	indirection operator	accesses the value at the address.

Address Of Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

Example:

```
#include<stdio.h>
int main(){
int number=50;
printf("value of number is %d, address of number is %u",number,&number);
return 0;
}
```

Output

```
value of number is 50, address of number is fff4
```

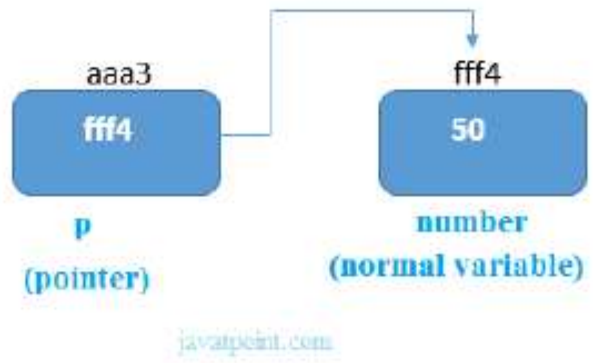
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol).

```
int *a;//pointer to int
char *c;//pointer to char
```

Pointer example

An example of using pointers printing the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable i.e. fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer **example** as explained for above figure.

```
#include<stdio.h>
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p);
printf("Value of p variable is %d \n",*p);
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

NULL Pointer

A pointer that is not assigned any value but NULL is known as NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will a better approach.

```
int *p=NULL;
```

In most the libraries, the value of pointer is 0 (zero).

Example: Pointer Program to swap 2 numbers without using 3rd variable

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

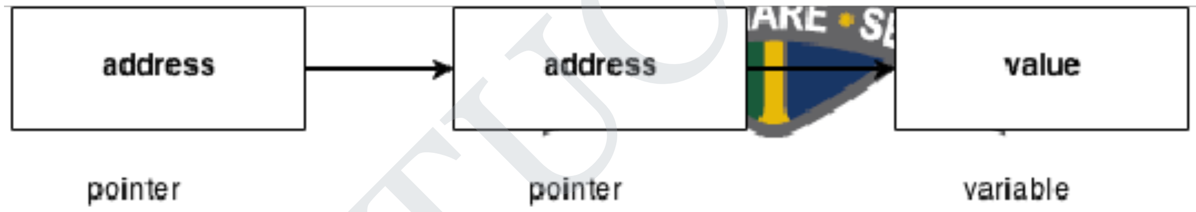
```
int main(){
int a=10,b=20,*p1=&a,*p2=&b;
 printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);
  return 0;
}
```

Output:

```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

3.6 POINTER TO POINTER

In C pointer to pointer concept, a pointer refers to the address of another pointer. In c language, a pointer can point to the address of another pointer which points to the address of a value. Let's understand it by the diagram given below:

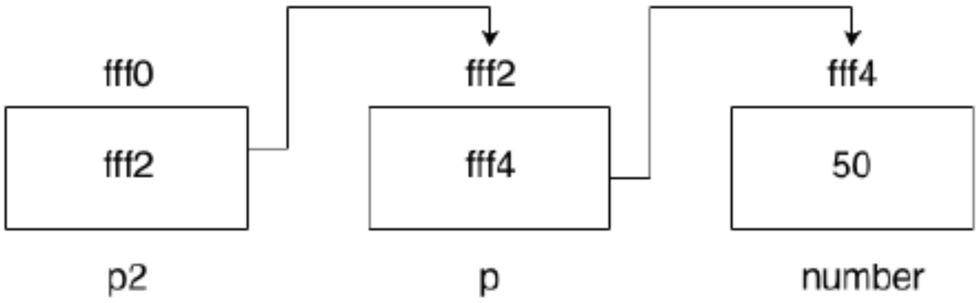


Let's see the syntax of pointer to pointer.

```
int **p2;
```

Example:

Let's see an example where one pointer points to the address of another pointer.



As you can see in the above figure, p2 contains the address of p (fff2) and p contains the address of number variable (fff4).

Example:

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
int **p2;//pointer to pointer
p=&number;//stores the address of number variable
p2=&p;
printf("Address of number variable is %x \n",&number);
printf("Address of p variable is %x \n",p);
printf("Value of *p variable is %d \n",*p);
printf("Address of p2 variable is %x \n",p2);
printf("Value of **p2 variable is %d \n",*p);
return 0;
}
```

Output:

```
Address of number variable is fff4
Address of p variable is fff4
Value of *p variable is 50
Address of p2 variable is fff2
Value of **p variable is 50
```

3.7 POINTER ARITHMETIC

In C pointer holds address of a value, so there can be arithmetic operations on the pointer variable. Following arithmetic operations are possible on pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction

- Comparison

Incrementing Pointer in C

Incrementing a pointer is used in array because it is contiguous memory location. Moreover, we know the value of next location.

Increment operation depends on the data type of the pointer variable. The formula of incrementing pointer is given below:

```
new_address= current_address + i * size_of(data type)
```

For 32 bit int variable, it will increment to 2 byte.

For 64 bit int variable, it will increment to 4 byte.

Let's see the example of incrementing pointer variable on 64 bit OS.

Example:

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p);
return 0;
}
```

Output:

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. The formula of decrementing pointer is given below:

```
new_address= current_address - i * size_of(data type)
```

For 32 bit int variable, it will decrement to 2 byte.

For 64 bit int variable, it will decrement to 4 byte.

Let's see the example of decrementing pointer variable on 64 bit OS.

Example:

```
#include <stdio.h>
void main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
printf("After decrement: Address of p variable is %u \n",p);
}
```

Output:

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

```
new_address= current_address + (number * size_of(data type))
```

For 32 bit int variable, it will add 2 * number.

For 64 bit int variable, it will add 4 * number.

Let's see the example of adding value to pointer variable on 64 bit OS.

Example:

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

Output:

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

As you can see, address of p is 3214864300. But after adding 3 with p variable, it is 3214864312 i.e. 4*3=12 increment. Since we are using 64 bit OS, it increments 12. But if we were using 32 bit OS, it were incrementing to 6 only i.e. 2*3=6. As integer value occupies 2 byte memory in 32 bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. The formula of subtracting value from pointer variable is given below:

```
new_address= current_address - (number * size_of(data type))
```

For 32 bit int variable, it will subtract 2 * number.

For 64 bit int variable, it will subtract 4 * number.

Let's see the example of subtracting value from pointer variable on 64 bit OS.

Example:

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-3; //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n",p);
return 0;
}
```

Output:

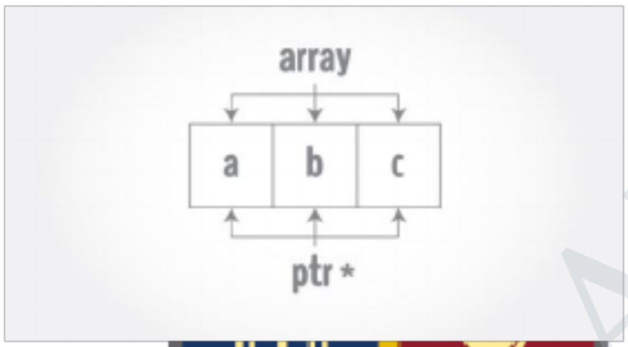
```
Address of p variable is 3214864300
```

```
After subtracting 3: Address of p variable is 3214864288
```

You can see after subtracting 3 from pointer variable, it is 12 (4*3) less than the previous address value.

3.8 ARRAYS AND POINTERS

Arrays are closely related to pointers in C programming but the important difference between them is that, a pointer variable takes different addresses as value whereas, in case of array it is fixed.



This can be demonstrated by an example:

```
#include <stdio.h>
int main()
{
   char charArr[4];
   int i;
   for(i = 0; i < 4; ++i)
   {
      printf("Address of charArr[%d] = %u\n", i, &charArr[i]);
   }
   return 0;
}
```

When you run the program, the output will be:

```
Address of charArr[0] = 28ff44
Address of charArr[1] = 28ff45
Address of charArr[2] = 28ff46
Address of charArr[3] = 28ff47
```

Note: You may get different address of an array.

Notice, that there is an equal difference (difference of 1 byte) between any two consecutive elements of array charArr.

But, since pointers just point at the location of another variable, it can store any address.

Relation between Arrays and Pointers

Consider an array:

int arr[4];

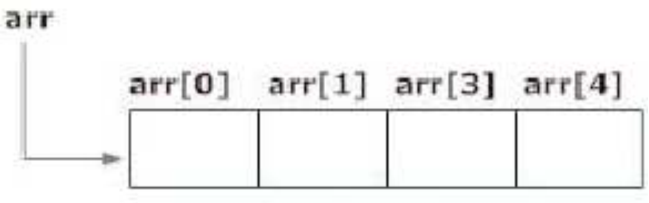


Figure: Array as Pointer

In C programming, name of the array always points to address of the first element of an array.

In the above example, arr and &arr[0] points to the address of the first element.

&arr[0] is equivalent to arr

Since, the addresses of both are the same, the values of arr and &arr[0] are also the same.

arr[0] is equivalent to *arr (value of an address of the pointer)

Similarly,

&arr[1] is equivalent to (arr + 1) AND, arr[1] is equivalent to *(arr + 1).

&arr[2] is equivalent to (arr + 2) AND, arr[2] is equivalent to *(arr + 2).

&arr[3] is equivalent to (arr + 3) AND, arr[3] is equivalent to *(arr + 3).

.

.

&arr[i] is equivalent to (arr + i) AND, arr[i] is equivalent to *(arr + i).

In C, you can declare an array and can use pointer to alter the data of an array.

Example: Program to find the sum of six numbers with arrays and pointers

```
#include <stdio.h>
int main()
{
 int i, classes[6],sum = 0;
 printf("Enter 6 numbers:\n");
 for(i = 0; i < 6; ++i)
 {
    // (classes + i) is equivalent to &classes[i]
    scanf("%d",(classes + i));
    // *(classes + i) is equivalent to classes[i]
    sum += *(classes + i);
 }
 printf("Sum = %d", sum);
 return 0;
}
```

Output:

```
Enter 6 numbers:
2
3
4
5
3
4
Sum = 21
```

3.9 ARRAY OF POINTERS

An array of pointers would be an **array that holds memory locations**. Such a construction is often necessary in the C programming language. Remember that an array of pointers is really an array of strings.

Example:

```
#include <stdio.h>
const int ARRAY_SIZE = 5;
int main ()
{
  /* first, declare and set an array of five integers:    */
  int array_of_integers[] = {5, 10, 20, 40, 80};
  /* next, declare an array of five pointers-to-integers: */
  int i, *array_of_pointers[ARRAY_SIZE];
   for ( i = 0; i < ARRAY_SIZE; i++)
  {
    /* for indices 1 through 5, set a pointer to
       point to a corresponding integer:              */
    array_of_pointers[i] = &array_of_integers[i];
  }
  for ( i = 0; i < ARRAY_SIZE; i++)
  {
    /* print the values of the integers pointed to
       by the pointers:                          */
    printf("array_of_integers[%d] = %d\n", i, *array_of_pointers[i] );
  }
  return 0;
}
```

Output:

```
array_of_integers[0] = 5
array_of_integers[1] = 10
array_of_integers[2] = 20
array_of_integers[3] = 40
array_of_integers[4] = 80
```

UNIT IV STRUCTURES

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

4.1 STRUCTURE

Structure in c language is a user defined datatype that **allows you to hold different type of elements.**

Each **element** of a structure is called a member.

It works like a template in C++ and class in Java. You can have different type of elements in it.

It is widely used to store student information, employee information, product information, book information etc.

Defining structure

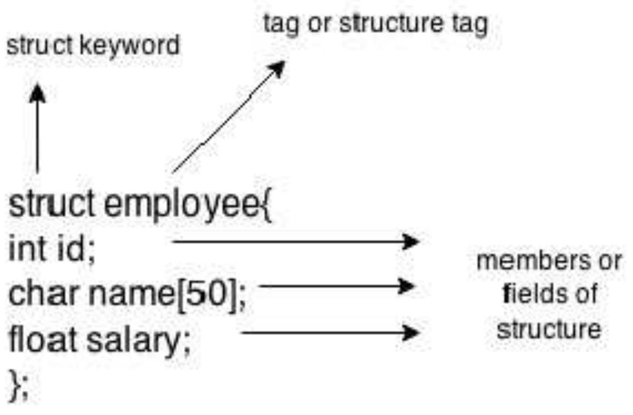
The **struct** keyword is used to define structure. Let's see the syntax to define structure in c.

```
struct structure_name
{
data_type member1;
data_type member2;
.
.
data_type memberN;
};
```

Let's see the **example** to define structure for employee in c.

```
struct employee
{   int id;
    char name[50];
    float salary;
};
```

Here, **struct** is the keyword, **employee** is the tag name of structure; **id**, **name** and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:



Declaring structure variable

We can declare variable for the structure, so that we can access the member of structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring variable at the time of defining structure.

1st way:

Let's see the example to declare structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{   int id;
   char name[50];
   float salary;
};
```

Now write given code inside the main() function.

```
struct employee e1, e2;
```

2nd way:

Let's see another way to declare variable at the time of defining structure.

```
struct employee
{   int id;
   char name[50];
   float salary;
}e1,e2;
```

Which approach is good

But if no. of variable are not fixed, use 1st approach. It provides you flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare variable in main() fuction.

Accessing members of structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by . (member) operator.

p1.id

Example:

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for structure
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
```

4.2 NESTED STRUCTURES

Nested structure in C language can have **another structure as a member**. There are two ways to define nested structure in c language:

1. By separate structure
2. By Embedded structure

Separate structure

We can create 2 structures, but dependent structure should be used inside the main structure as a member. Let's see the code of nested structure.

```
struct Date
{
  int dd;
  int mm;
  int yyyy;
};
struct Employee
{
  int id;
  char name[20];
  struct Date doj;
}emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

Embedded structure

We can define structure within the structure also. It requires less code than previous way. But it can't be used in many structures.

```
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
    int dd;
    int mm;
    int yyyy;
   }doj;
}emp1;
```

Accessing Nested Structure

We can access the member of nested structure by Outer_Structure. Nested_Structure.member as given below:

```
e1.doj.dd
e1.doj.mm
e1.doj.yyyy
```

4.3 ARRAY OF STRUCTURES

There can be array of structures in C programming to store many information of different data types. The array of structures is also known as collection of structures.

Let's see an example of structure with array that stores information of 5 students and prints it.

```
#include<stdio.h>
#include <string.h>
struct student{
int rollno;
char name[10];
};
int main(){
int i;
struct student st[5];
printf("Enter Records of 5 students");
for(i=0;i<5;i++){
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",&st[i].name);
}
printf("\nStudent Information List:");
for(i=0;i<5;i++){
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
  return 0;
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz
Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

4.4 DYNAMIC MEMORY ALLOCATION

The concept of dynamic memory allocation in c language **enables the C programmer to allocate memory at runtime.** Dynamic memory allocation in c language is possible by 4 functions of **stdlib.h** header file.

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

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

Static memory allocation	**Dynamic memory allocation**
Memory is allocated at compile time.	Memory is allocated at run time.
Memory can't be increased while executing program.	Memory can be increased while executing program.
Used in array.	Used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	Allocates single block of requested memory.
calloc()	Allocates multiple block of requested memory.
realloc()	Reallocates the memory occupied by malloc() or calloc() functions.
free()	Frees the dynamically allocated memory.

malloc()

The malloc() function **allocates single block of requested memory** .

It doesn't initialize memory at execution time, so it has **garbage value initially**.

It returns NULL if memory is not sufficient.

The **syntax** of malloc() function is given below:

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

Example:

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
      printf("Sorry! unable to allocate memory");
      exit(0);
```

```
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output:

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

calloc()

The calloc() function allocates **multiple block of requested memory**.

It initially initialize **all bytes to zero.**

It returns NULL if memory is not sufficient.

The **syntax** of calloc() function is given below:

```
ptr=(cast-type*)calloc(number, byte-size)
```

Example:

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output:

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc()

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the **syntax** of realloc() function.

```
ptr=realloc(ptr, new-size)
```

free()

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the **syntax** of free() function.

```
free(ptr)
```

4.5 SELF REFERENTIAL STRUCTURES

A self referential structure is used to create data structures like linked lists, stacks, etc. Following is an example of this kind of structure:

```
struct struct_name
{
datatype datatypename;
struct_name * pointer_name;
};
```

A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type. For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. For example,

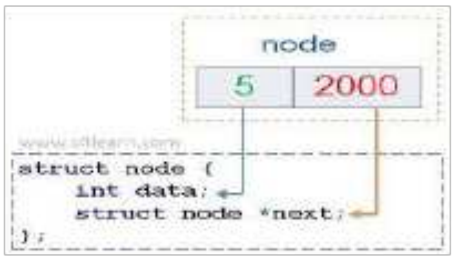
```
typedef struct listnode {
void *data;
struct listnode *next;
} linked_list;
```

In the above example, the listnode is a self-referential structure – because the *next is of the type struct listnode.

4.6 SINGLY LINKED LIST

A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

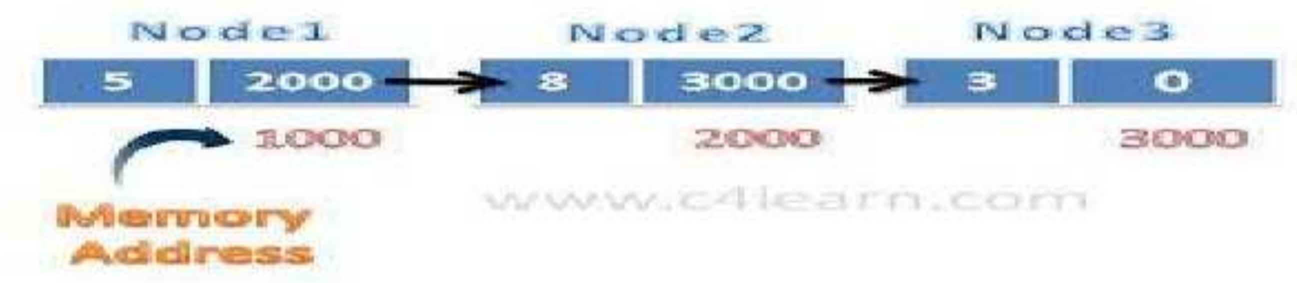
Node:



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

A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

Linked List:



A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

Declaring a Linked list :

In C language, a linked list can be implemented using structure and pointers .

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

The above definition is used to create every node in the list. The **data** field stores the element and the **next** is a pointer to store the address of the next node.

In place of a data type, **struct LinkedList** is written before next. That's because its a **self-referencing pointer**. It means a pointer that points to whatever it is a part of. Here **next** is a part of a node and it will point to the next node.

Creating a Node:

Let's define a data type of struct LinkedList to make code cleaner.

```
typedef struct LinkedList *node; //Define node as pointer of data type struct LinkedList
node createNode(){
    node temp; // declare a node
    temp = (node)malloc(sizeof(struct LinkedList)); // allocate memory using malloc()
    temp->next = NULL;// make next point to NULL
    return temp;//return the new node
}
```

typedef is used to define a data type in C.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file stdlib.h.

sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to malloc.

The above code will create a node with data as value and next pointing to NULL.

Let's see how to **add a node to the linked list**:

```
node addNode(node head, int value){
    node temp,p;// declare two nodes temp and p
    temp = createNode();//createNode will return a new node with data = value and next
pointing to NULL.
    temp->data = value; // add element's value to data part of node
    if(head == NULL){
        head = temp;     //when linked list is empty
    }
    else{
        p  = head;//assign head to p
        while(p->next != NULL){
            p = p->next;//traverse the list until p is the last node.The last node always points
to NULL.
        }
        p->next = temp;//Point the previous last node to the new node created.
    }
    return head;
}
```

Here the new node will always be added after the last node. This is known as **inserting a node at the rear end**.

Food for thought

This type of linked list is known as simple or singly linked list. A simple linked list can be traversed in only one direction from head to the last node.

The last node is checked by the condition :

```
p->next = NULL;
```

Here -> is used to access **next** sub element of node p. **NULL** denotes no node exists after the current node , i.e. its the end of the list.

Traversing the list:

The linked list can be traversed in a while loop by using the **head** node as a starting reference:

node p;

```
p = head;
while(p != NULL)
{
   p = p->next;
}
```

4.7 TYPEDEF

The C programming language provides a keyword called **typedef**, by using this keyword you can create a user defined name for existing data type. Generally typedef are use to create an **alias name** (nickname).

Declaration of typedef

```
typedef datatype alias_name;
```

Example:

```
typedef int tindata;
```

Example program:

```
#include<stdio.h>
#include<conio.h>
typedef int intdata;
void main()
{
int a=10;
integerdata b=20
typedef intdata integerdata;//Intergerdata is again alias name of intdata
integerdata s;
s=a+b;
printf("\nSum::%d",s);
getch();
}
```

Output:

```
Sum::30
```

Code Explanation

- In above program Intdata is an user defined name or alias name for an integer data type.
- All properties of the integer will be applied on Intdata also.
- Integerdata is an alias name to existing user defined name called Intdata.

Advantages of typedef

- It makes the program more portable.
- Typedef make complex declaration easier to understand.

typedef with struct

Take a look at below structure declaration

```
        struct student{
        int id;
        char *name;
        float percentage;
        };
struct student a,b;
```

As we can see we have to include keyword struct every time you declare a new variable, but if we use typedef then the declaration will as easy as below.

```
        typedef struct{
        int id;
        char *name;
        float percentage;
        }student;
```

student a,b;

This way typedef make your declaration simpler.

UNIT V FILE PROCESSING

Files – Types of file processing: Sequential access, Random access – Sequential access file - Example Program: Finding average of numbers stored in sequential access file - Random access file - Example Program: Transaction processing using random access files – Command line arguments

5.1 FILES

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for **permanent storage of data**. It is a readymade structure.

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will **preserve your data** even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can **easily access the contents** of the file using few commands in C.
- You can **easily move your data from one computer** to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the **normal .txt files** that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as **plain text**. You can easily edit or delete the contents.

They take minimum effort to maintain, are **easily readable**, and provide least security and **takes bigger storage space**.

2. Binary files

Binary files are mostly the **.bin files** in your computer.

Instead of storing data in plain text, they **store it in the binary form (0's and 1's).**

They can hold higher amount of data, are not readable easily and provides a **better security than text files**.

File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file
5. C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	**description**
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the beginning point

Opening a File or Creating a File

The **fopen()** function is used to create a new file or to open an existing file.

Syntax:

```
*fp = FILE *fopen(const char *filename, const char *mode);
```

Here, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file.

filename is the name of the file to be opened and **mode** specifies the purpose of opening the file. Mode can be of following types,

Mode	Description
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

Closing a File

The fclose() function is used to close an already opened file.

Syntax :

```
int fclose( FILE *fp);
```

Here fclose() function closes the file and returns **zero** on success, or **EOF** if there is an error in closing the file. This **EOF** is a constant defined in the header file **stdio.h**.

Input/ Output operation on File

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are the simplest functions which can be used to read and write individual characters to a file.

Example:

```
#include<stdio.h>
int main()
{
    FILE *fp;
    char ch;
    fp = fopen("one.txt", "w");
    printf("Enter data...");
    while( (ch = getchar()) != EOF) {
        putc(ch, fp);
    }
    fclose(fp);
    fp = fopen("one.txt", "r");
     while( (ch = getc(fp)! = EOF)
    printf("%c",ch);
        // closing the file pointer
    fclose(fp);
        return 0;
}
```

Reading and Writing to File using fprintf() and fscanf()

```
#include<stdio.h>
struct emp
{
    char name[10];
    int age;
};
void main()
{
    struct emp e;
    FILE *p,*q;
    p = fopen("one.txt", "a");
    q = fopen("one.txt", "r");
    printf("Enter Name and Age:");
    scanf("%s %d", e.name, &e.age);
    fprintf(p,"%s %d", e.name, e.age);
    fclose(p);
    do
    {
        fscanf(q,"%s %d", e.name, e.age);
        printf("%s %d", e.name, e.age);
    }
    while(!feof(q));
}
```

In this program, we have created two FILE pointers and both are refering to the same file but in different modes.

fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on the console using standard printf() function.

Difference between Append and Write Mode

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already.

The only difference they have is, when you **open** a file in the **write** mode, the file is reset, resulting in deletion of any data already present in the file. While in **append** mode this will not happen.

Append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the **cursor is positioned at the end of the present data** in the file.

Reading and Writing in a Binary File

A Binary file is similar to a text file, but it contains only large numerical data. The Opening modes are mentioned in the table for opening modes above.

fread() and fwrite() functions are used to read and write is a binary file.

Syntax for writing a binary file:

```
fwrite(data-element-to-be-written, size_of_elements, number_of_elements, pointer-to-file);
```

fread() is also used in the same way, with the same arguments like fwrite() function. Below mentioned is a simple example of writing into a binary file

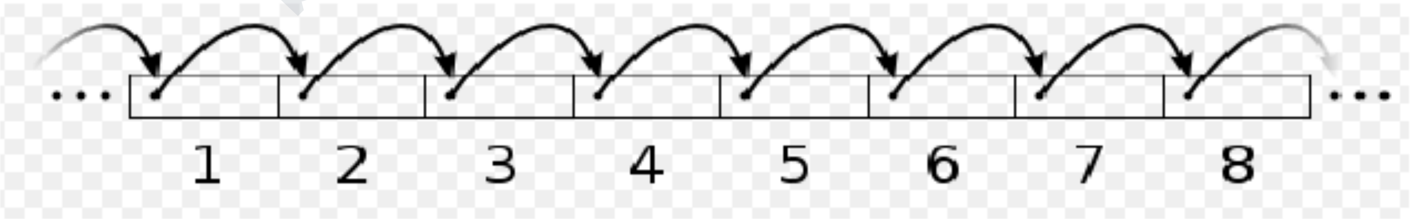
```
const char *mytext = "The quick brown fox jumps over the lazy dog";
FILE *bfp= fopen("test.txt", "wb");
if (bfp)
{
   fwrite(mytext, sizeof(char), strlen(mytext), bfp);
   fclose(bfp);
}
```

5.2 TYPES OF FILE PROCESSING: SEQUENTIAL ACCESS, RANDOM ACCESS

In computer programming, the two main types of file handling are:

- **Sequential access**

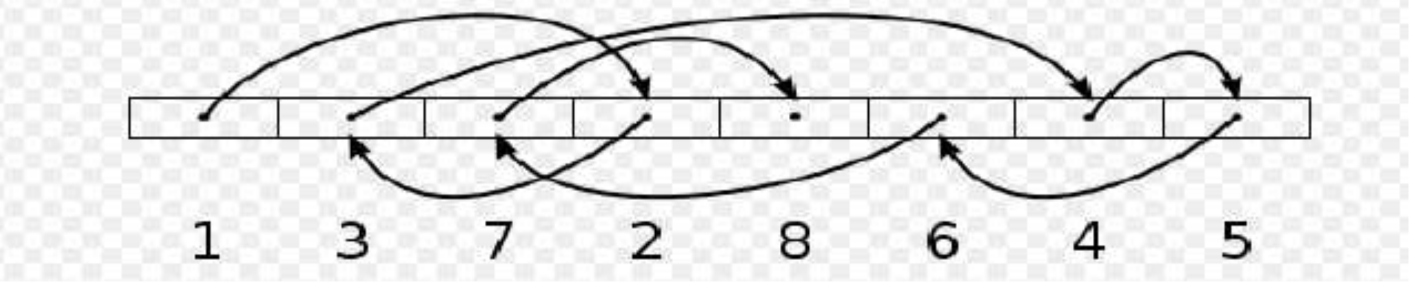
  In this type of files data is kept in sequential order if we want to read the last record of the file, we need to read all records before that record so it takes more time.



Sequential access to file

- **Random access**

  In this type of files data can be read and modified randomly .If we want to read the last record we can read it directly. It takes less time when compared to sequential file.



Random Access To File

There is no need to read each record sequentially, if we want to access a particular record. C supports these functions for random access file processing.

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

fseek():

It is used to **move the reading control to different positions** using fseek function.

Syntax:

```
fseek( file pointer, displacement, pointer position);
```

Where

file pointer ---- It is the pointer which points to the file.

displacement ---- It is positive or negative. This is the number of bytes which are skipped backward (if negative) or forward(if positive) from the current position. This is attached with L because this is a long integer.

Pointer position:

This sets the pointer position in the file.

Value	Pointer Position
0	Beginning of file
1	Current position
2	End of file

Example:

1) fseek(p,10L,0)

0 means pointer position is on beginning of the file, from this statement pointer position is skipped 10 bytes from the beginning of the file.

2)fseek(p,5L,1)

1 means current position of the pointer position. From this statement pointer position is skipped 5 bytes forward from the current position.

3)fseek(p,-5L,1)

From this statement pointer position is skipped 5 bytes backward from the current position.

ftell(): It tells the **byte location of current position of cursor** in file pointer.

rewind(): It **moves the control to beginning** of the file.

Example program for fseek():

Write a program to read last ‘n’ characters of the file using appropriate file functions(Here we need fseek() and fgetc())

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char ch;
clrscr();
fp=fopen("file1.c", "r");
if(fp==NULL)
printf("file cannot be opened");
else
{
printf("Enter value of n  to read last ‘n’ characters");
scanf("%d",&n);
fseek(fp,-n,2);
while((ch=fgetc(fp))!=EOF)
{
printf("%c\t",ch);}
}
}
```

```
fclose(fp);
getch();
}
```

5.3 COMMAND LINE ARGUMENTS

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax:

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

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example:

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
{
  int i;
  if( argc >= 2 )
  {
    printf("The arguments supplied are:\n");
    for(i = 1; i < argc; i++)
    {
      printf("%s\t", argv[i]);
    }
  }
  else
  {
    printf("argument list is empty.\n");
  }
  return 0;
}
```

Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be 1.