

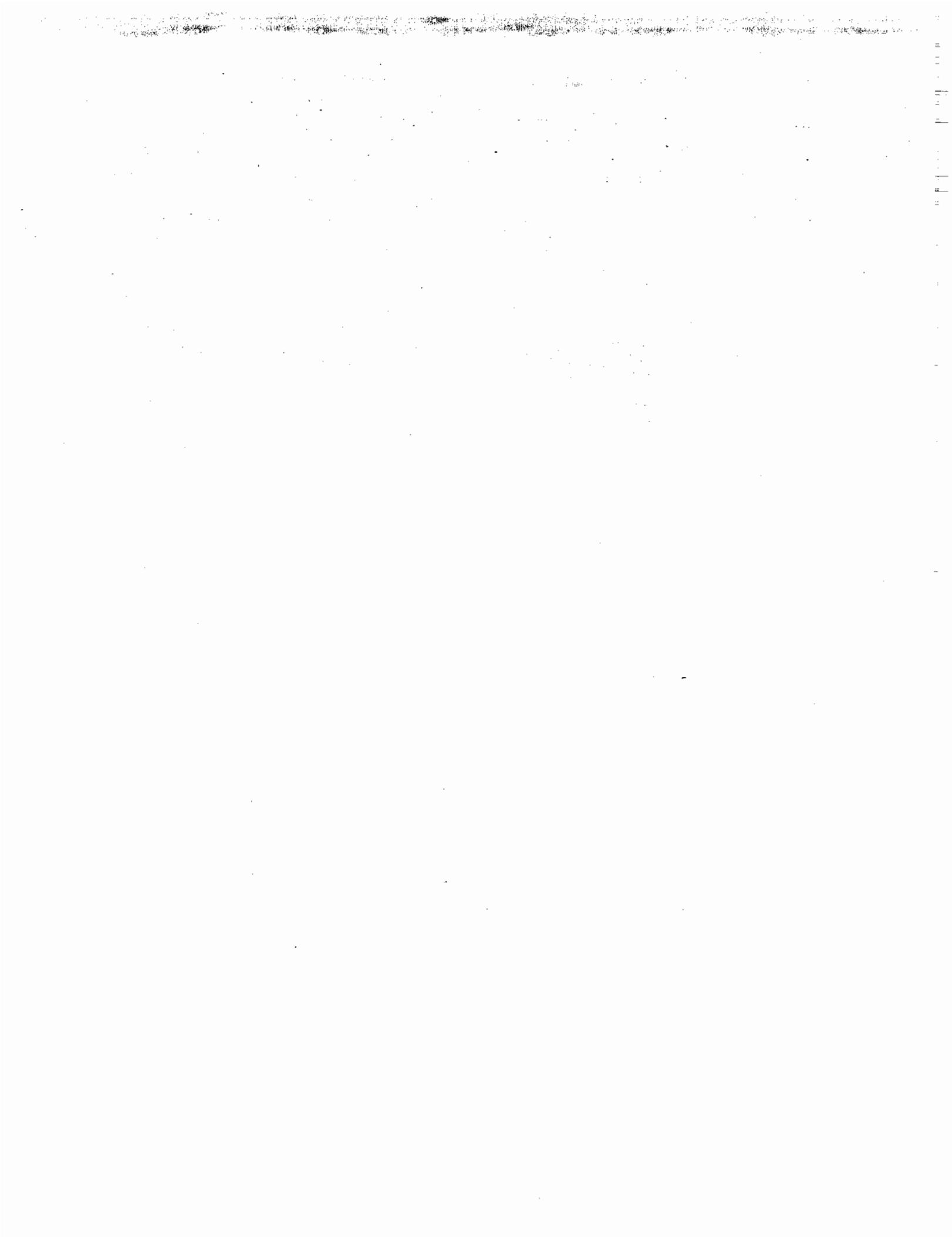
# **C++ NOTES**

**BY**  
**SATHISH GUPTHA**

**NARESH TECHNOLOGIES**

**MANOJ XEROX**

Gayarti nager,Behind-Huda-Ameer pet  
SOFT WARE INSTITUTES MATERIAL AVAILABLE  
CELL :8978989710



AS

## Introduction

Q @ what is C++ ?

Q A C++ is an object oriented programming language.  
But object oriented is not a language. It is  
programming principle.

Q @ What is programming principle ?

Q A A programming principle define set of rules  
and regulations. By following this principle  
complexity and developing program can be reduced.

Q Programming elements :-

Q Programming elements are two types.

Q Those are 1. data

Q 2. Instructions

Q Every program requires input, process and  
output.

Q Input :-

Q The data or information given to the  
program is called as Input.

Q Process :-

Q Programming some calculation is  
called as process.

Q Output :-

Q It is a result.

Q What is program?

A A program is set of data and instructions.

Q What is programming?

A Programming is organizing of data and instructions according to given problem.

List of programming principles :-

There are six types of programming principles. These are as follows.

1. unstructured programming (USP)

2. procedural oriented programming (POP)

3. Modular (or) Structured oriented programming

4. object oriented programming (OOP)

5. component Based programming (CBP)

6. Aspect oriented programming (AOP)

1. unstructured programming (USP) :-

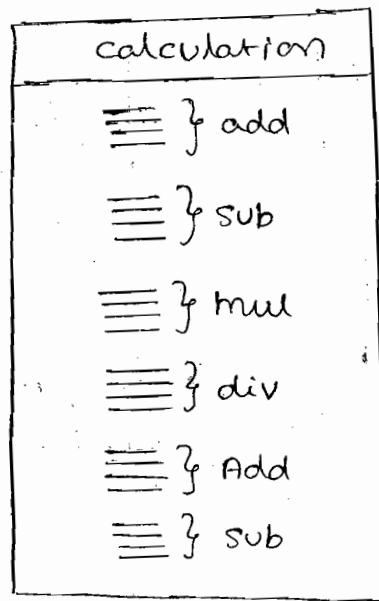
According to unstructured programming concept, program is developed by organizing data and instructions in sequential order. It is organizing sequential order.

Drawbacks of unstructured programming :-

\* Redundent (Repetition) code/ instructions

\* Because of Redundency The size of program increased.

- \* occupy more space, which reduce speed.
- \* There is no proper organization of data and operations.



Ex:-

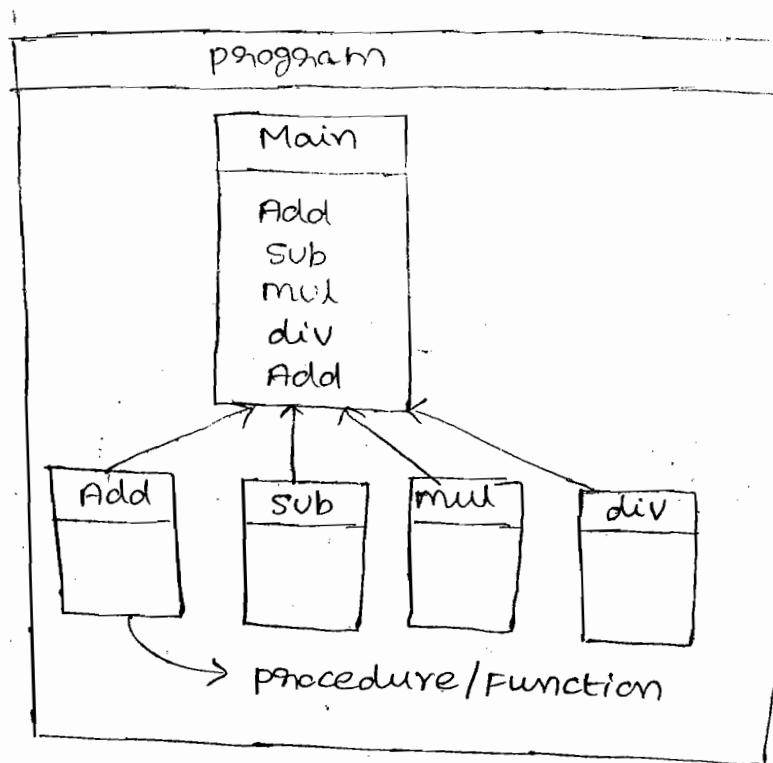
Assembly Language, machine Language are unstructured programming Languages.

## 2. Procedural oriented programming (POP) :-

operations provided by a program is divided into small pieces and each piece of a program is called SubRoutine.

Q. what is SubRoutine?

A. a small program with in a program is called SubRoutine. The SubRoutine can be either procedure or function.



Ex:-

COBAL and PASCAL are called procedural oriented languages.

Advantages of procedural oriented programming :-

These are four types.

1. modularity.
2. Reusability.
3. simplicity.
4. Efficiency.

1. modularity :-

Dividing programming instructions

According to its operations into small modules.

2. Reusability :-

write once and use many times.

### 3. Simplicity :-

It is easy to understand operations of a program.

### 4. Efficiency :-

By Reducing The size of The program

Efficiency of procedure oriented programs increases.

### Drawbacks of procedural oriented programming :-

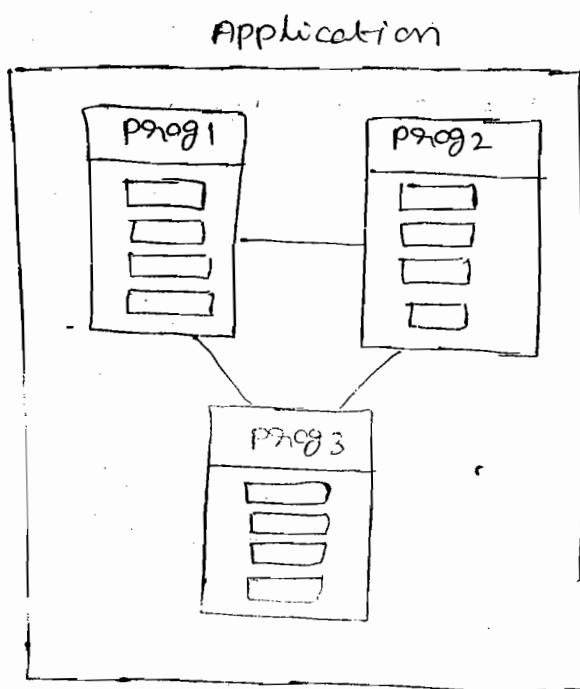
- \* concentrated on development of functions.
- \* Little attention is given to the data being used by function.
- \* In a Large program it is very difficult to identify what data is used by which function.
- \* when revising an External data, structure all the functions that Access the data also has to be Revised
- \* Debugging Application is an complex.
- \* There is no proper organization of data and operations.

### Characteristics of procedural oriented programming

- \* Emphasis is on doing things (algorithms)
- \* Large programs are divided into small programs.
- \* most of the functions share global data

- \* Data moves openly around the system.
- \* Functions transforms data from one to another.
- \* Employs top-down approach.

### 3. Structured oriented programming (or) modular oriented programming Language C Sop (or) Mop :-



- \* Modular oriented program is a superset of procedural oriented programming approach.
- \* In this approach reusability of subroutines are between the programs.
- \* The drawbacks of procedural can be applicable to modular oriented programming Language.
- \* 'C' is a structured oriented (or) procedural oriented programming Language

#### 4. Object oriented programming :-

Q) Name some pure object oriented Languages ?

A) Smalltalk,  
Java,  
Eiffel,  
Sather.

Q) why C++ is not a pure object oriented Language ?

A) It does not fulfill all the principles of object oriented.

C++ is an advanced 'C' language. The programs developed in 'C' language with or without modifications moved to C++.

#### Object oriented concepts :-

1. Encapsulation :-

\* Encapsulation is a process of grouping data and related operations.

\* Wrapping up of data and code into a single unit (class).

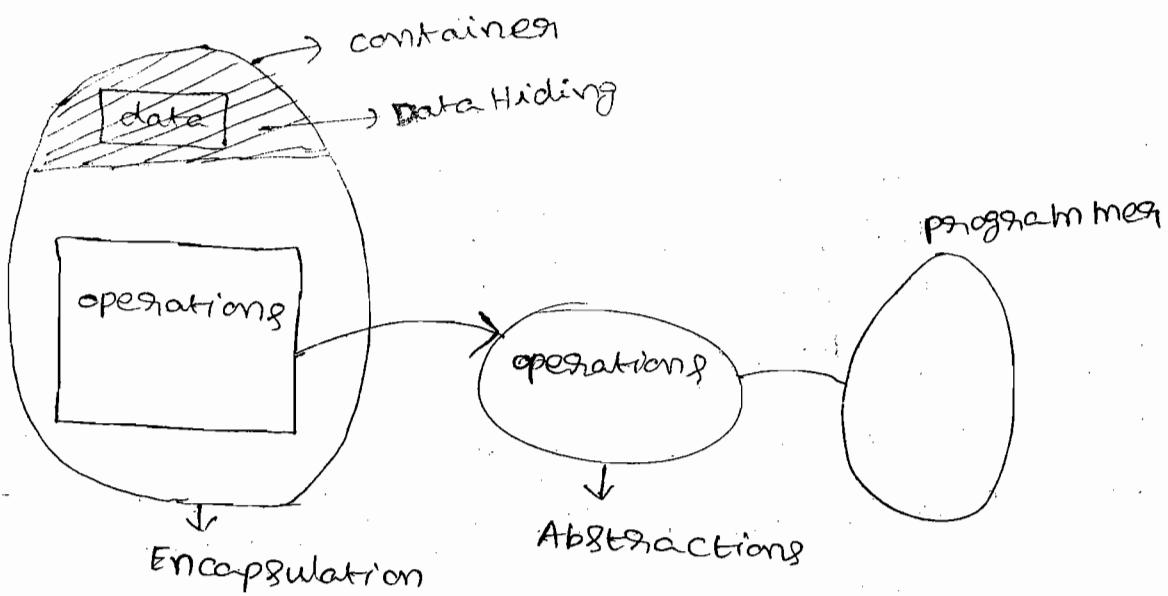
\* Data is not accessible by the outside world.

\* Only the functions wrapped up in the class can access it.

#### Data Hiding :-

\* The insulation of the data from direct access from the program.

- \* Helps the programmers to build secure programs.



Abstraction :

Abstraction is a process by which River required operations

Encapsulation: It is a process of organizing data and related operations in proper order

Advantages :

- ① Hiding
- ② Binding

characteristics of object oriented programming :-

\* Emphasis is on data rather than procedure

\* Programs are divided into objects.

\* Function that operate on the data are tied together in the data structure.

\* Data is hidden and can not be accessed by external functions.

- \* objects may communicate with each other.
- \* New data and functions may be easily added.
- \* employs bottom-up approach.

class :-

- \* class is a container which contains data and operations.
- \* class is Encapsulated with information and instructions.
- \* class is a data type/user defined datatype.
- \* class is a blueprint of object.
- \* blueprint is nothing but a planning before constructing an object.
- \* class is a logical view of object.
- \* class is planning before constructing object.
- \* class is a collection of members
  1. data members.
  2. member functions.
- \* class is a collection of variables and functions.
- \* An user defined data type, the entire set of data and code to manipulate data.
- \* objects are variables of type class.
- \* Any number of objects of a class can be created.

## object :-

- \* object is an instance of class.
- \* object is a class variable.
- \* object is a physical representation of a class.

(Q) what is an SBI of object ?

(A) SBI stands for State Behaviour and Identity.  
Every object has above 3 properties

### state :-

A value given to variable/attribute of object

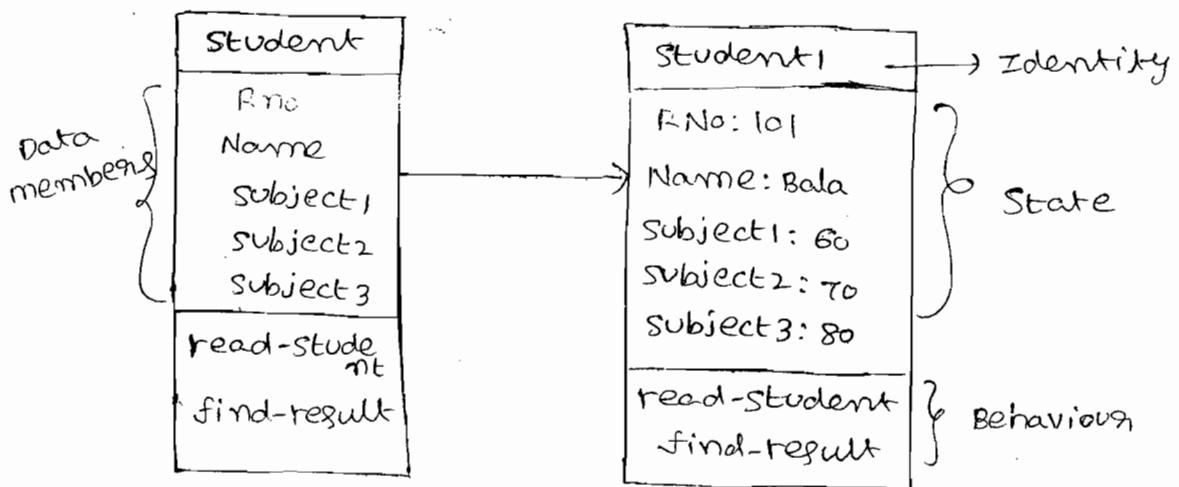
define state.

### Behaviour :-

An operation which change the state of object or manipulate contents of object.

### Identity :-

A unique name to identify each object.



## C++ → to Generalized

- \* C++ is an object oriented programming Language.
- \* C++ is superset of C Language.
- \* The idea comes from operators.
- \* C++ is an advanced "C" Language

### why C++ ?

- \* object oriented programming Language.
- \* C++ is a portable Language

### Portable :-

- \* C++ programs can be developed on different operating systems.
- \* C++ programs are portable across multiple operating systems.
- \* Portable means write once and more programs to change any operating systems.
- \* C++ is a platform dependent because when C++ program is ~~compiled~~ the compiler generates platform dependent code

↓  
O.S

### C compatibility :-

The programs which are developed in 'C' without any modification moved to C++

- \* C++ is an Embedded programming language which is having the characteristics of Low level and high level Languages.

## Applications of C++ :-

### Application Software :-

I can develop Application Softwares are

- Database (Oracle)
- wordprocessor (MS-word)
- Spread sheet (MS-Excel)
- Editors (Notepad, wordpad)

A computer which solve the problem of End user,  
is called Application software.

### System software :-

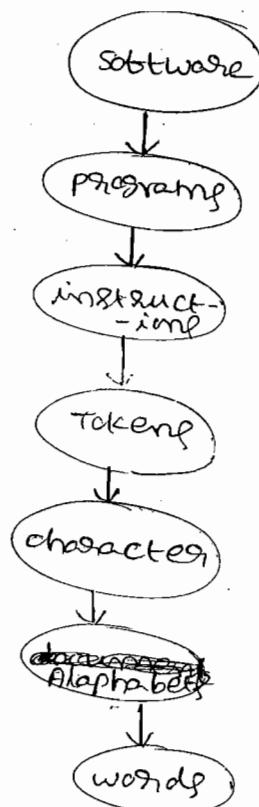
- operating system.
- compilers.
- devices drivers
  - ↳ printer
  - ↳ mouse
  - ↳ keyboard,
  - ↳ monitor.

### Business Application :-

- Bank
- supermarket
- Hospital
- Students school / college

### communication :-

\* Protocols (HTTP) most of telecommunication is developed in C++.



## History of C++ :-

C++ to 80's

- \* An object-oriented programming Language.
  - \* Initially named C with classes.
  - \* Developed by Bjarne Stroustrup.
  - \* Renamed as C++ in 1993.
  - \* The idea comes from C increment operator ++
  - \* It is a superset of C.
- 
- \* A software is a collection of programs, a program is a set of instructions.
  - \* An instruction is a collection of tokens.
  - \* A token is collection of characters.

## Character Set of C++ :-

\* C++ uses ASCII code character set.

\* According to ANSI, no. of characters used in C++ for developing programs is 128.

A - Z

a - z

0 - 9

Special characters are

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

## Tokens of C++ :-

C++ is a case sensitive language. It finds difference between capital letters and small letters.

A token is a smallest individual unit with in the program. A program is never developed without tokens.

There are 5 types of tokens those are

1. keywords
2. Identifiers.
3. operators.
4. Datatypes.
5. constants.

### 1. Keywords :-

Keyword is a language related word. having specific meaning with in language. meaning of this word can not be changed.

bool keyword :- There a keyword called bool. It is a datatype.

class, template, virtual, friend, public, private, protected, namespace, try, catch, throw, new, delete, operator These are keywords in C++.

True, false

\* In C++ The total no of keywords is 48.

\* In C The total no of keywords are 32

## Data types :-

These are four types of Data types.

① Fundamental Data types.

② Derived data types.

③ user defined data types.

④ Empty data types.

① Fundamental Data types :-

1. int —— 2 bytes.

2. unsigned int —— 2 bytes.

3. signed int —— 2 bytes.

4. short int —— 2 bytes.

5. long int —— 4 bytes

6. float —— 4 bytes

7. double —— 8 bytes

8. long double —— 10 bytes.

9. char —— 1 byte

10. unsigned char —— 1 byte

11. signed char —— 1 byte

12. bool —— 1 byte.

Signed means Accepting the positive and negative values. Signed & unsigned int & unsigned int are same.

② Derived data type:-

① Array

② Pointers

③ Reference.

### ⑧ User defined datatype

~~~~~ ~~~~~

int a=1;

1. struct.

2. union.

3. Enum.

4. class.

### ⑨ Empty Data type

~~~~~ ~~~~~

Void.

### operators in C++

~~~~~ ~~~~~

The operators which are added in  
C++ are

1. new

2. delete

3. :: (Scope Resolution operator)

4. \* (Pointer to member Access operator)

5. \*:: (pointer to member Access operator)

### constants :-

boolean constants

i) true.

ii) false

Ex:- int x; float y; bool b;

x=10; y=1.5; b=true;

### ⑩ differences between cout and printf

cout

1. It is an object
2. formatted and unformatted
3. It doesn't specify formatted specification

printf

1. It is a function
2. formatted output
3. It required formatting specification

## I/o streams :-

### Stream :-

- \* A Stream is a class or object which provide input and output operations.
- \* A Stream is a flow of bytes having source or destination.
- \* In c++ i/o operations are performed using two predefined objects.

(i) cin (ii) cout

### cout :- (console output)

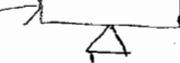
- \* cout is an object of type ostream class.
- \* cout performs formatted/unformatted output operations.
- \* cout does not use formatting specifiers.
- \* formatting is an implicit functionality of cout.
- \* This object is available or declare within I/o stream.h (iostream.h).
- \* This object is bind with an operator << (insertion operator), in order to insert data within output stream.
- \* insertion operation is a function available in cout object.
- \* cout object represents monitor.
- \* It is an overloaded left shift operator.
- \* The behaviour of this operator changes when used with fundamental data types.

Syntax :-

`cout << Variable/constant;`

Ex :-

`cout << "Hello";` → 

`cout << 10;` → 

\* How to print more than one value using "cout"?

(A) cascading insertion operators.

\* Using more than one insertion operation with cout is called cascading.

Ex :-

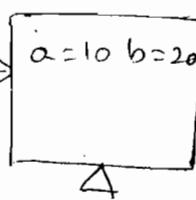
`a=10 b=20`

`cout << a << b;` → 

`cout << "a=" << a;` → 

~~cout~~

\* The Evaluation of insertion operator are done from right to left and printing is done from left to right

`cout << "a=" << a << "b=" << b;` → 

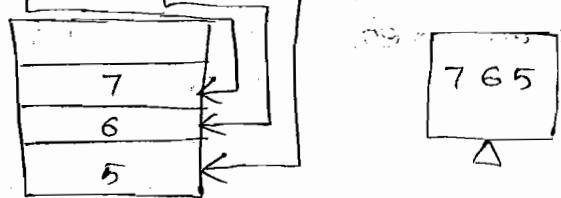
Ex :- `int a;`

`a=5;`

`cout << a++ << a++ << a++;`

←

`cout << a + << a + << a;`



Ex :-

`int a;`

`a = 5;`

`cout << a + << a + << a;`

`a = 5`

`a = 6`

`a = 6`

`a = 7`

`a = 7`

`a = 8`



Ex :-

`int a;`

`a = 5;`

`a = 5`

`a++ = 6`

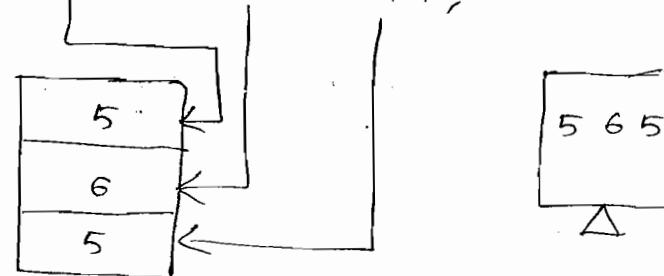
`a = 6`

`a-- = 5`

`a = 5`

`a++ =`

`cout << a + << a - << a;`



`cin :- (Console input)`

\* `cin` is an object of `istream` class.

\* `cin` represents standard input device (Keyboard)

\* `cin` uses `>>` (extraction operator)

\* which extracts data from keyboard and stores inside memory.

\* It is an overloaded operator.

Syntax :-

cin >> variable;

\* It perform formatted/unformatted input operations.

\* It does not uses formatting specifiers.

cin >> var1 >> var2 >> var3 ...

C++ compilers :-

C++ compilers available in market are

Turbo C++

Borland C++

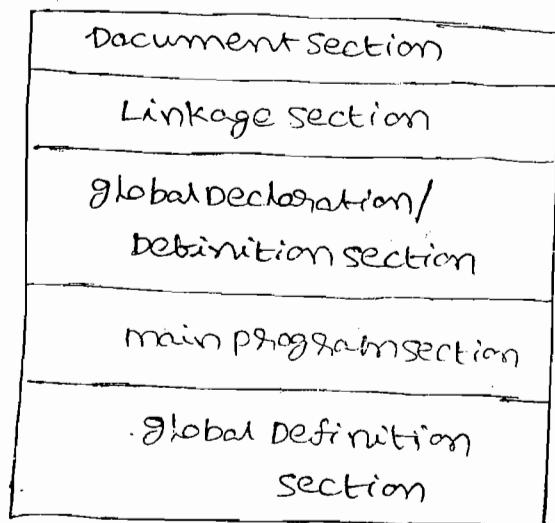
Microsoft C++

C++ on unix

C++ on Linux

C++ on Solaris.

Structure of C++ program :-



structure of C++ program can be divided into five type.

1. Document section.
2. Linkage section.
3. Global Declaration / Definition section.
4. Main program section.
5. Global Definition section.

#### 1. Document Section :-

\* It is <sup>an</sup> optional section.

\* This section consists information about program.

\* It is not understand by compiler.

\* This documentation is provided using comments.

\* C++ provides two types of comments.

1. singletline comment (//)

2. multiline comment /\* \*/

\* comments are ignored by compiler.

Ex:- int rno; // room no

/\*

this is logic for finding even or odd \*/

if (num % 2 == 0)

#### 2. Linkage Section (or) preprocessor section :-

\*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\*

\* This section is mandatory section.

\* This section is executed before compiling  
Linking

The program and it is used for other program  
with ~~the~~ current program

\*\*\* command used for preprocessor directives is

# include

\*\* Q what is difference between # include <filename>  
and # include "filename".

A

# include <filename>

1. Search file name in include path.

# include "filename"

1. Search file name in source path if not found search in include path.

③ Global Declaration or Definition section :-

\* use for Declaring global Variables and data types and functions.

\* It is an optional section.

\* This is against object oriented principle because do it c++ it's not true object oriented.

④

Main program section :-

\* Every Executable c++ program must have the function main

\* Execution of c++ program starts from main function

Syntax - 1

int main ()  
{

==

\* return 0 :-

0 (zero) is an exit control value return  
to operating system to terminate the execut  
ion of the program.

return 1 :-

1 (one) is operating system does not  
execute the program it gives the error.

Syntax - ii :-

empty data type

Void main ( )

{

====

}

\* The libraries doesn't have the main program.

\* It is an optional section.

⑤ Global Definition Section :-

\* Writing the function and data type  
definition which have declared in global  
declaration section

\* It is an optional section.

\* Turbo C++ 3.0 doesn't support Exception handling  
and name space and inline functions.

Program :-

```
/* This is my first program */

#include <iostream.h>

int main ()
{
    cout << "welcome to c++";
    return 0;
}
```

Save program :-

c++ program is save by .cpp extension

\* Turbo C++ 3.0 include other program and current program and compiles.

\* Turbo C++ 4.5 does not include file but it navigates from current program to included program

compiling the program → Alt+F9

Run the program → Ctrl+F9

Q) what is compiler, linker and loader?

A)

Compiler :-

which translates source program into object program.

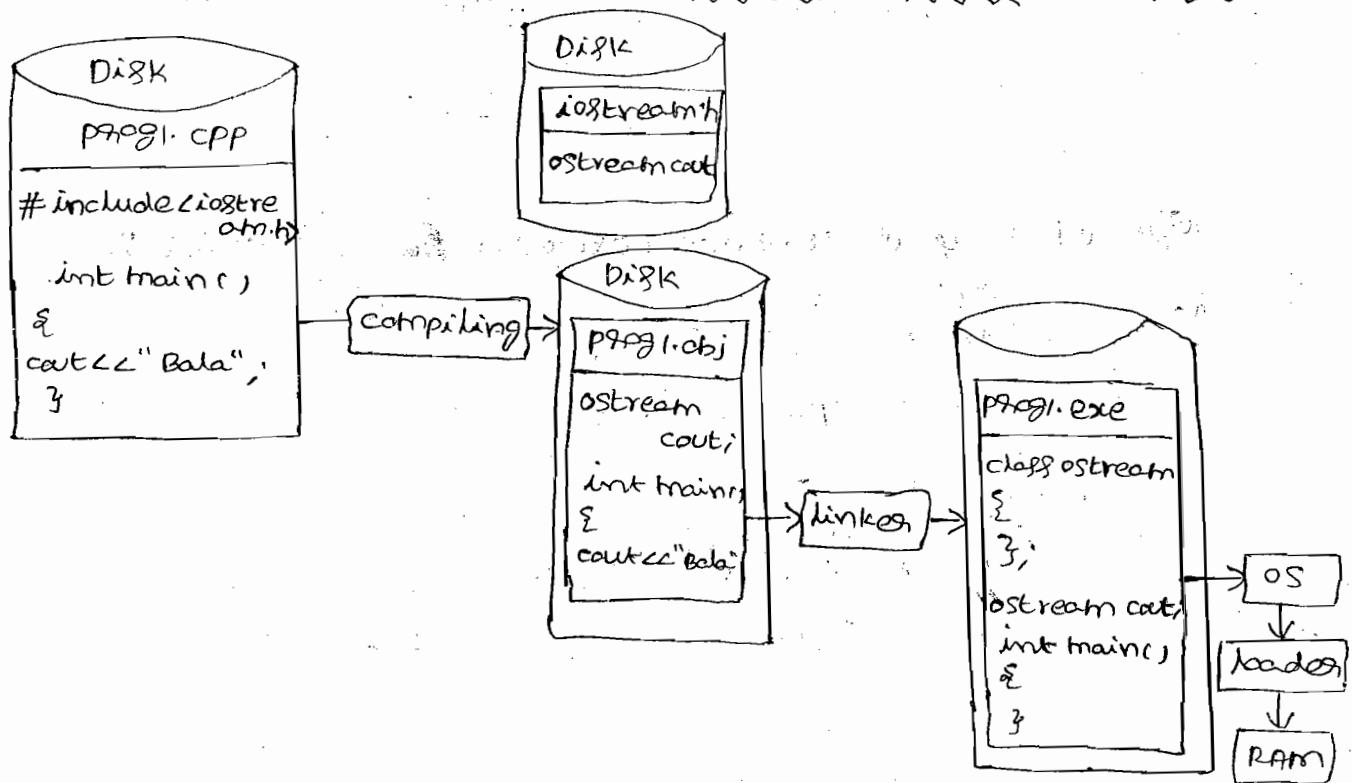
Linker :-

which translates object program into executable program.

Loader :-

provided by operating system,

Secondary storage to main memory (Primary storage)  
 Compiling and linking and Execution of C++ program :-



Program :-

```

// working cout object

#include <iostream.h>
#include <conio.h>

int main()
{
    clrscr();
    cout << 10 << endl;
    cout << 1.5 << endl;
    // cout << true << endl;
    cout << "C++" << endl;
    return 0;
}
  
```

O/P  
10  
1.5  
C++

Q) What is endl?

A) The above program display compile time error

because All the compilers does not support boolean values true, false

Q) what is difference between \n and endl ?

A) → \n :-  
→ \n is called Escape sequence.

→ It occupy one byte or space.

endl :-

→ It is a manipulator

→ It does not occupy ~~any~~ ~~to~~ any space.

The functionality of \n and endl is same because both inserting one newline.

Local Variables :-

~~~~ ~~~~

Q) what is variable?

A) → Variable is a container, which contains value  
→ Variable is a named memory location which holds value.

Q) what is Local Variable?

A) \* A variable declared inside function is called local variable

\* C++ allows to declare local variables anywhere within the functions.

\* Program main area can be divided into 4 types

- Data Area.
- Heap Area.
- Stack Area.
- Text Area.

Characteristics of Local Variables:-

→ Type :-

datatype which is define type or value hold by variable.

→ Scope :- The scope of local variable is with in function.

→ initial value :-

garbage/unknown

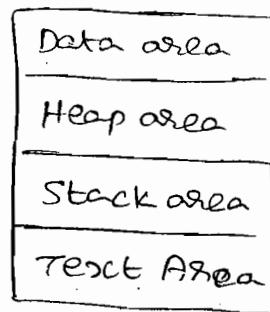
→ lifetime :-

untill the execution function.

→ storage :-

Stack.

\* Every program the operating system reserve the memory by dividing into four blocks.



\* The code which has to be process is loaded inside text area.

Syntax :-

datatype Variable-name, ---;

~~program :-~~

\* Local variables are called automatic variable (or) auto variable

\* auto is a storage for a class.

Program :-

// local variables.

```
#include <iostream.h>
#include <conio.h>
int main ()
{
    int x, y;
    clrscr();
    cout << x << endl << y;
    return 0;
}
```

O/P:- garbage variables.

## Initialization:-

QUESTION 1

\* declaring a variable by assigning value

is called initialization.

\* C++ supports two types of initializations

1. Static initialization.

2. Dynamic initialization.

Ex :-

int x; → It is declaration.

int x=10; → It is Initialization.

### Static initialization:-

\* Declaring a variable by assigning constant value is called of static initialization

Syntax:-

datatype variablename = constant value;

Ex:- int x=10;

\* A value assigned to the ~~variable~~ variable known by compiler is called static initialization.

### Dynamic initialization:-

→ A values which are assigned to the variable not known by the compiler is called dynamic initialization.

→ In this type variable is assigned Expression.

Program :-

// Finding area of triangle

```
#include <iostream.h>
```

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

```
int main ()
```

```
{
```

```
float base = 1.5, height = 2.5; → static
```

```
cout << "In Base" << base; initialization
```

```
cout << "In Height" << height;
```

```
float area = 0.5 * base * height; → dynamic
```

```
cout << "In Area of triangle" << area; initialization
```

```
return 0;
```

```
}
```

Program :-

// Write a program to read name, three  
subject marks and calculate total and  
Average.

```
#include <iostream.h>
```

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

```
int main ()
```

```
{
```

```
char name[10];
```

```
int sub1, sub2, sub3;
```

```
clrscr();
```

O/P

Base 1.5

Height 2.5

Area 1.875

```

cout << "Input name",
      cin >> name;

cout << "Input 3 subject marks",
      cin >> sub1 >> sub2 >> sub3;

int total = sub1 + sub2 + sub3;
float avg = total / 3.0;

cout << "Name" << name;
cout << "Total." << total;
cout << "Avg" << avg;
return 0;
}

```

\* Extraction operator(`>>`) can read only one word in string but can not read more than one word in string.

\* `cout` performs by default unformatted operations

Manipulators :-

\* causes a specified operation to be performed on a stream.

\* used to format the data display.

\* most commonly used manipulators are `setw` and `endl`

\* `setprecision()`, `setfill()`, `setiosflags()`, are other manipulators.

\* Manipulator is a function executed by ~~cout~~ to perform ~~cout~~ formatted output operations.

\* These functions are available in iomanip.h

Block Level Variables:

C++ supports different types of blocks

1. Function block.
2. Conditional block.
3. Iterational block.
4. Anonimous block.

\* Variables declared inside a function block are called local variables.

\* The variables declared inside the body of the block are called block level variables.

\* C++ supports three types of variables global variable and local variable and block level variables.

Characteristics of block level Variables:

- Scope :- with in block.

- lifespan :- until the block is executed.

- initial value :- garbage

- Storage :- stack.

Ex:-

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

int main()
{
    char name[20];
    int sub1, sub2, sub3;
    clrscr();
    cout << "\n input name";
    cin >> name;
    cout << "\n input 3 subject marks";
    cin >> sub1 >> sub2 >> sub3;
    cout << "\n Name " << name;
    if (sub1 < 40 || sub2 < 40 || sub3 < 40)
        cout << "\n Result Fail";
    else
    {
        int total = sub1 + sub2 + sub3;
        float avg = total / 3.0;
        cout << "\n Total " << total;
        cout << "\n Avg " << avg;
        cout << "\n Result pass";
    }
    return 0;
}
```

\* C++ Allows to declare local and block level variable with same name. which is called as hiding.

Ex :-

```
int main ()  
{  
    int x=10;  
    cout << x; → 10  
  
    {  
        int x=30;  
        cout << x; → 30  
    }  
    cout << x; → 10  
    return 0;  
}
```

Anonymous  
block

Ex :-

```
int main ()  
{  
    int x=10, y=20;  
    cout << x << y; → 10 20  
  
    {  
        int y=40;  
        cout << y; → 40  
        cout << x; → 10  
        y = y+10 → 50  
    }  
    cout << x; → 10  
    cout << y; → 20
```

## NUMBER SYSTEM

1. Decimal Number System.
2. Binary Number System.
3. Octal Number System.
4. Hexadecimal Number System.

### ① Decimal Number System :-

An integer value with base 10 is called decimal Integer which consists of digits from 0 to 9

Ex:  $(123)_{10}$

$$\begin{aligned} &= 10^0 \times 3 + 10^1 \times 2 + 10^2 \times 1 \\ &= 1 \times 3 + 10 \times 2 + 100 \times 1 \\ &= 3 + 20 + 100 \\ &= 123 \end{aligned}$$

Decimal number:

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

Binary number:

000 ← zero  
001 ← one  
010 ← Two  
011 ← Three  
100 ← Four  
101 ← Five

Ex:-  $(729)_{10}$

$$\begin{aligned} &= 9 \times 10^0 + 2 \times 10^1 + 7 \times 10^2 \\ &= 9 \times 1 + 2 \times 10 + 7 \times 100 \\ &= 9 + 20 + 700 \\ &= 729 \end{aligned}$$

Binary Number system :-

An Integer value with base 2 is called  
Binary Number System. It consists of 0's and 1's

Ex

$$(111101)_2$$

$$\begin{aligned} &= 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 \\ &= 1 \times 1 + 0 \times 2 + 1 \times 4 + 1 \times 8 + 1 \times 16 + 1 \times 32 \\ &= 1 + 0 + 4 + 8 + 16 + 32 \\ &= 61 \end{aligned}$$

Convert decimal to Binary :-

To convert  $(123)_{10}$  to Binary

$$\begin{array}{r} 123 \\ 2 | \hline 61 - 1 \\ 2 | \hline 30 - 1 \\ 2 | \hline 15 - 0 \\ 2 | \hline 7 - 1 \\ 2 | \hline 3 - 1 \\ \hline 1 - 1 \end{array}$$

$$(123)_{10} = (1111011)_2$$

Convert Binary to Decimal :-

$$(1111011)_2$$

$$= 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 1 \times 2^6$$

$$= 1 \times 1 + 1 \times 2 + 0 \times 4 + 1 \times 8 + 1 \times 16 + 1 \times 32 + 1 \times 64$$

$$= 1 + 2 + 0 + 8 + 16 + 32 + 64$$

$$= (123)_{10}$$

## Octal Number System :

- \* An Integer value with base 8 is called octal Integer
- \* It consist of digit from 0 to 7
- \* octal integer is prefix with zero.

Ex:  $(126)_8$

$$\begin{aligned}
 &= 8^0 \times 6 + 8^1 \times 2 + 8^2 \times 1 + 8^3 \times 0 \\
 &= 1 \times 6 + 8 \times 2 + 64 \times 1 + 0 \\
 &= 6 + 16 + 64 \\
 &= 86
 \end{aligned}$$

This conversion is not direct. First we convert Decimal to octal

$$(65)_{10} = (101)_8$$

$$\begin{array}{r}
 8 \overline{)65} \\
 8 \overline{)8} - 1 \\
 1 - 0
 \end{array}
 \quad = 1 \times 8^0 + 0 \times 8^1 + 1 \times 8^2 \\
 \quad = 1 + 0 + 64 \\
 \quad = (65)_{10}$$

Ex :-

~~126~~  
10

$$(126)_{10} \text{ convert } \underline{\text{_____}}$$

$$\begin{array}{r}
 10 \overline{)126} \\
 10 \overline{)2} - 1 \\
 1 - 0
 \end{array}
 \quad = 6 \times 10^0 + 2 \times 10^1 + 1 \times 10^2 \\
 \quad = 6 + 20 + 100 \\
 \quad = \underline{\underline{126}}$$

Ex:  $(73)_{10}$  convert to octal

$$\begin{array}{r} 8 \overline{)73} \\ 8 \overline{)9} - 1 \\ 1 - 1 \end{array}$$
$$= 1 \times 8^0 + 1 \times 8^1 + 1 \times 8^2$$
$$= 1 + 8 + 64$$
$$(73)_{10} = (111)_8$$
$$\underline{\underline{= (73)_{10}}}$$

Hexadecimal Number system :

\* An Integer value with base 16 is called Hexa decimal integer.

\* It consists of digits from 0-9 and A-F

\* This integer is prefix with 0x

0 1 2 3 4 5 6 7 8 9 A B C D E F

Ex:

$$(65)_{10} = (0x41)_{16} \quad (126)_{10} = (0x7E)_{16}$$
$$16 \overline{)65} \quad 16 \overline{)126}$$
$$4 - 1 \quad 7 - 14$$

Hexa decimal to decimal :

$$(0x7E)_{16}$$
$$= 16^0 \times E + 16^1 \times 7$$
$$= 16^0 \times 14 + 16^1 \times 7$$
$$= 1 \times 14 + 112$$
$$= (126)_{10}$$
$$\underline{\underline{=}}$$

Decimal to Hexa decimal :

$$(1220)_{10} = (4C4)_{16}$$

$$\begin{array}{r} 16 \longdiv{1220} \\ 16 \quad \boxed{76-4} \\ 16 \quad \boxed{4-12} \\ \hline & \quad 0-4 \end{array}$$

Binary fraction to decimal fraction :

$$(11010.11)_2$$

$$\begin{aligned} &= 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 1 \times 2^{-2} + 1 \times 2^{-1} \\ &= 0 + 2 + 0 + 8 + 16 + 0.25 + 0.5 \\ &= 26.75 \end{aligned}$$

Decimal to Binary :

$$(19)_{10} = (10011)_2$$

$$\begin{array}{r} 2 \longdiv{19} \\ 2 \quad \boxed{9-1} \\ 2 \quad \boxed{4-1} \\ 2 \quad \boxed{2-0} \\ \hline & \quad 1-0 \end{array}$$

Decimal to octal :

$$(89)_{10} = (131)_8$$

$$\begin{array}{r} 8 \longdiv{89} \\ 8 \quad \boxed{11-1} \\ 8 \quad \boxed{1-3} \\ \hline & \quad 0-1 \end{array}$$

Decimal fraction to binary fraction :

$$(19.75)_{10}$$

$$\begin{array}{r} 19 \\ \hline 2 | 9-1 \\ 2 | 4-1 \\ 2 | 2-0 \\ \hline & 1-0 \end{array}$$

$$0.75 \times 2 = 1.50 \rightarrow 1$$

$$0.5 \times 2 = 1.0 \rightarrow 1$$

$$0.0 \times 2 = 0$$

$$(19.75)_{10} = (10011.11)_2$$

- \* If local and block level variable with same name local variable can't access with in block.

Global Variables :- (or) External Variable :-

- \* The variables which are declared at the top of the program and outside the function is called global variables. These variables are not secured.
- \* C++ is not a pure object oriented because it supports global variables which is against Encapsulation.

Ex:- ~~#include <iostream.h>~~

# include <conio.h>

int x; → global variable

Characteristics of Global Variables :-

Scope :-

with in program

Lifetime :-

until program is executed

Storage :-

Data type

Initial Value :-

int - 0

float - 0.0

char - null (non-printable)

↓  
'\0'

character

\* C++ allows to declare local and global variable with same name

Ex:-

```
#include <iostream.h>
#include <conio.h>
int x=10;           → global variable
int Main()
{
    int y=20;       → local variable
    cout << x; → 10
    cout << y; → 20
    return 0;
}
```

Ex:-

```
#include <iostream.h>
#include <conio.h>
int x=10;
int main()
{
    int y=20, x=30;
    cout << x; → 30
    cout << y; → 20
    cout << ::x → 10
    return 0;
}
```

Scope resolution operators (::) :-

If local and global variables with same name global variable is referred with in function using scope resolution operators (::)

② Find the output

int x=10; → global variable

int main()

{

    int x=20; → local variable

    cout << x; → 20

}

    int x=30; → block level variable

    cout << x; → 30

    cout << ::x; → 10

}

Output :- 20 30 10

② Find output ?

int x=20;

int main()

{

    int y=30;

    cout << ::x; → 20

    cout << y; → 30

~~return 0;~~

    cout << x; → 20

    return 0;

}

Output : 20 30 20

## Manipulators :-

### setw( ) :-

→ when used with numericf they will be displayed right justified.

Ex:- cout << setw(5) << sum << endl;

output : 123

Ex:-

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
```

O/P

10  
200  
3000

int main ( )

{

```
int x=10, y=200, z=3000;
clrscr();
cout << setw(5) << x << endl;
cout << setw(5) << y << endl;
cout << setw(5) << z << endl;
return 0;
```

}

### setfill( ) :-

~~~~~

→ In the display the unused positions of the field are filled with white spaces, by default.

→ By using setfill( ), we can fill the unused positions with the desired character.

### Syntax :-

setfill("\*");

Ex :-

```
cout << setfill('*') << setw(10) << 5250 << endl;
```

output : \*\*\*\*\*5250

~~Setprecision()~~

~~~~~ ~~~~

→ By default, the floating numbers are printed with digits after the decimal point.

→ we can specify the number of digits to be displayed after the decimal point.

→ This can be done by using the setprecision() manipulator

Ex :- cout << setprecision(3);

float x = 12.12345

cout << "x = " << x ,

output : 12.123

② Find output ?

```
#include <iomanip.h>
#include <iostream.h>
#include <iostream.h>
int main()
{
    float a=10;
    cout << setprecision(2) << a;
    return 0;
}
```

output : 10

\* setprecision does not display precision until unless the value are same precision.

Q) Find output?

```
# include <iostream.h>
# include <conio.h>
# include <iomanip.h>

int main ()
{
    float a=1.2345, b=1.4567;
    clrscr();
    cout<< setprecision(2)<<a << endl;
    cout<< setprecision(3)<<b ;
    return 0;
}
```

| output |
|--------|
| 1.23   |
| 1.457  |

\* setprecision perform two operations  
(i) Truncating (ii) Rounding.

~~setiosflags()~~ :-

\* This is used for field justification e.g. left or right

Ex:- cout<< setiosflags(ios::right),  
cout<< setiosflags(ios::left);

\* used for displaying float field in fixed or scientific notation

Ex:- cout<< setiosflags(ios::fixed),  
cout<< setiosflags(ios::scientific),

\* Used for displaying trailing zeros in a float field.

Ex: cout << setiosflags(cios::showpoint);  
                    ↓  
            class

\* Used for displaying positive sign.

Ex: cout << setiosflags(cios::showpos);  
                    ↓  
            class      ↓  
                    DataManipulator

Program:

```
#include <iomanip.h>
#include <iostream.h>
#include <conio.h>
int main()
{
    int a=10, b=-20;
    clrscr();
    cout << setiosflags(cios::showpos) << a << endl;
    cout << setiosflags(cios::showpos) << b;
    return 0;
}
```

O/P  
+10  
-20

Program:

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
int main()
{
    float a=10;
    clrscr();
    cout << setiosflags(cios::showpoint) << a;
    return 0;
}
```

O/P  
10.00000

\* Show point displays for the precision of default numbers  
of precision (2.22, 1.5 etc.)

Program :-

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
int main()
{
    float a=1.5;
    clrscr();
    cout<<a<<endl;
    cout<<setiosflags(ios::scientific)<<a;
    return 0;
}
```

O/P  
1.5  
1.5e+00

→ setiosflags(ios::oct) : octal format

setiosflags(ios::hex) : hexa format

Note :-

octal and hexa used only with integer can't  
used with float.

Program :-

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
int main()
{
    int x=65;
    clrscr();
    cout<<x<<endl;
    cout<<setiosflags(ios::oct)<<x;
    cout<<setiosflags(ios::hex)<<x;
    return 0;
}
```

## Control Structures or Statements

These are divided into three types.

- ① conditional statements,
- ② un-conditional statements,
- ③ Looping statements.

Default flow of executing program is sequential. In this all the statements are processed one by one (sequential order). In order to change the flow of executing a program we use control statements.

### conditional statements :

- \* moving Execution control from one place to another place based on condition.
- \* There are two types.

- ① Decision making statement (if)
- ② Selection Statement (switch)

### if statement :

- \* moves the execution control based on return value of condition/ boolean Expression.

- \* There are 4 types.

- ① simple if.
- ② if-else.
- ③ if-else-if.
- ④ nested if.

① simple-if :-

~~~~~

\* if statement without else block.

\* if statement with only one block (true block)

Syntax:

~~~~~

(i) if (expression)

    statement;

(or)

(ii) if (expression)

{

    statement 1;

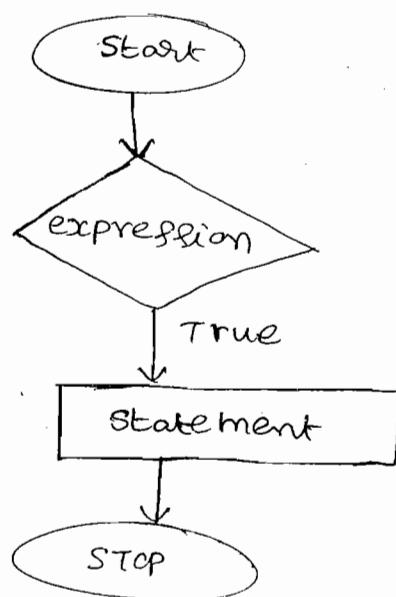
    statement 2;

-----

}

Flow chart:

~~~~~



Program :

write a program to read name, age, of person and Eligible to vote

```
#include <iostream.h>
#include <conio.h>
int main()
{
    char name[20];
    int age;
    clrscr();
    cout<<"\n Input Name:";
    cin>>name;
    cout<<"\n Input Age:";
    cin>>age;
    if (age >= 18)
        cout<<" Eligible for vote "<<"It" <<name;
    return 0;
}
```

Output :

Input Name : bala

Input Age : 22

Eligible for vote bala.

\* if statement can be terminated with semicolon(;) .

if does not have else block.

\* if is terminated with semicolon means if does not have any statement if executed the condition is true.

.. if (expression)

    Statement-1;

    Statement-2;

- \* Expression is true the two statements are Executed.

i.e. Statement-1;

    Statement-2;

- \* Expression is false the statement 2 is Executed.  
i.e Statement-2;

## ② if-else :-

- \* An if statement followed by another block (else)

Syntax:

~~~~~

if (expression)

    Statement1;

else

    Statement 2;

- \* If expression return non-zero (true) execute Statement 1.

- \* If expression return zero (false) execute Statement 2.

Syntax:

~~~~~ :

    if (expression,

{

        Statement 1;

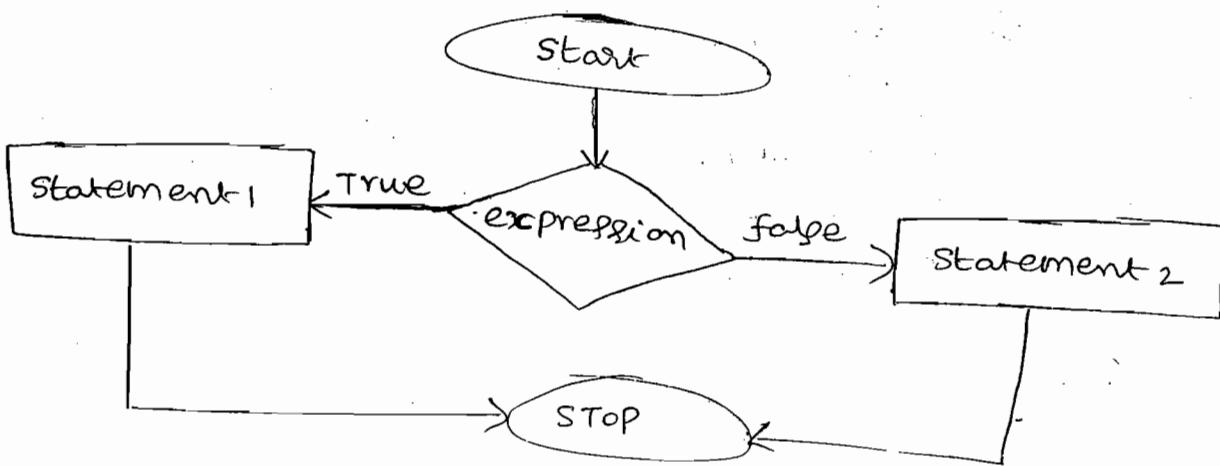
        Statement 2;

Statement 3;

Statement 4;

\* The above syntax is Executing multiple statements

Flow chart :



Program :

write a program to read a character  
and find vowel or not

```
#include <iostream.h>
```

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

```
int main()
```

```
{
```

```
char ch;
```

```
clrscr();
```

```
cout<<"\n Input Any character:";
```

```
cin>> ch;
```

```
if(ch=='A'||ch=='E'||ch=='I'||ch=='O'||  
ch=='U')
```

```
cout<<"\n vowel";
```

```
cout << "In not vowel",  
return 0;  
}
```

Output : Input Any character : A

Vowel

Input Any character : a  
not vowel.

if expression)

{

Statement 1,

Statement 2,

Statement 3,

}

\* if expression is true it executes statement one,  
two and three

\* If expression is false if statement 1 is skip and  
executes statement 2 and statement 3.

Program :

write a program to find given number is  
even or odd.

```
#include <iostream.h>
```

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

```
int main ()
```

{

int num;

clrscr();

```
cout << "Input Any Number:",
```

```
cin >> num;
```

```
if (num % 2 == 0)
    cout << "Even Number";
else
    cout << "odd Number";
return 0;
}
```

output : Input Any Number : 4

Even Number.

Input Any Number : 7

odd Number.

program :

write a program to find given year is leap year or not.

```
#include <iostream.h>
#include <conio.h>
int main ()
{
    int year,
    clrscr(),
    cout << "Input Any Year : ";
    cin >> year,
    if (year % 4 == 0)
        cout << " Leap Year",
    else
        cout << " not Leap Year",
    return 0;
}
```

output :

Input Year : 2008      Input Year : 2010

if-else-if ;  
~~~~~

\* To Evaluate the multiple condition to use if-else-if

Syntax:  
~~~~~

if (expression-1)

    Statement 1;

else

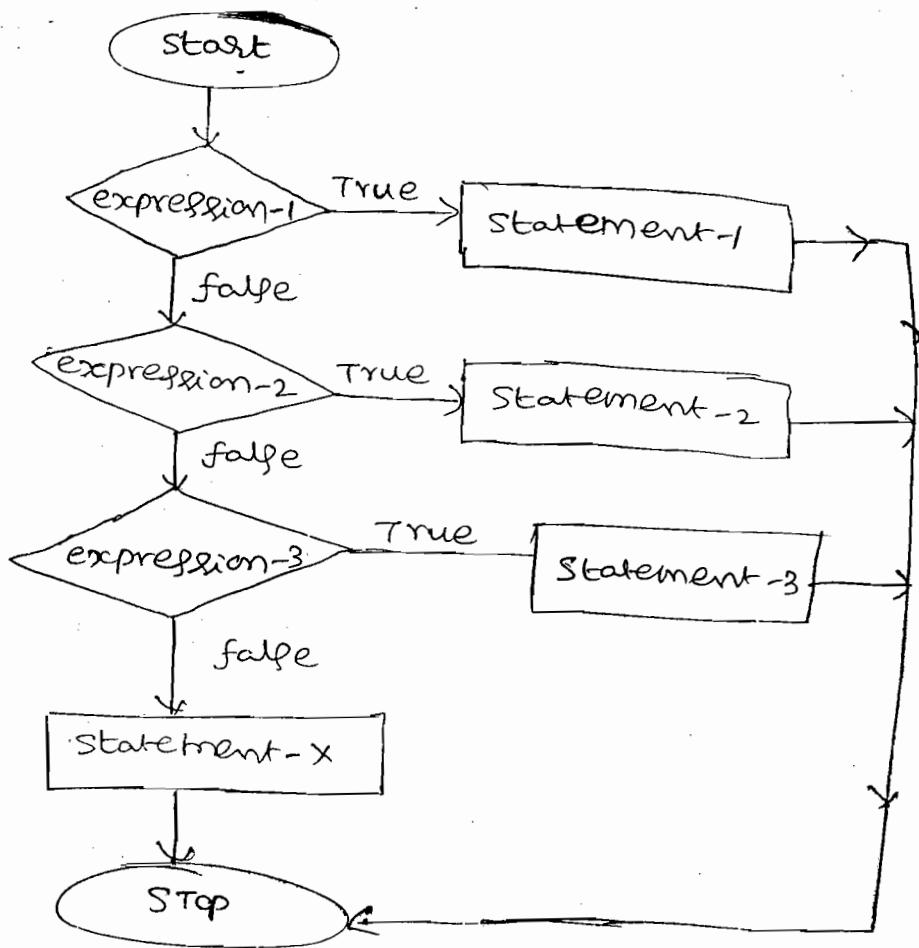
    if (expression-2)

        Statement 2;

    else

        Statement - x;

flowchart:  
~~~~~



if (expression-1)

    Statement 1;

else

if (expression-2)

    Statement 2;

else

if (expression-3)

    Statement 3;

else

    Statement x;

Program :

Write a program to find Max of Three  
Numbers.

```
#include <iostream.h>
```

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

```
int main()
```

```
{
```

```
    int a,b,c,
```

```
    clrscr();
```

```
cout<<"In Input a,b,c:";
```

```
cin>>a>>b>>c;
```

```
if (a>b &&a>c)
```

```
cout<<"In a is maximum:" <<a;
```

```
else
```

```
    if (b>c)
```

```
        cout<<"In b is maximum:" <<b;
```

```
    else
```

```
        cout<<"In c is maximum:" <<c;
```

```
    return 0;
```

```
}
```

out put : Input a,b,c values: 10 20 30

c is maximum: 30

Input a,b,c values: 10 5 2

a is maximum: 10

program :

write a program to find Input character is  
Alphabet, digit or special character.

```
#include<iostream.h>
#include<conio.h>
int main ()
{
    char ch;
    clrscr();
    cout << "Input Any character: ";
    cin >> ch;
    if ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'))
        cout << "In Alphabet",
    else
        if ((ch >= '0' && ch <= '9'))
            cout << "In digit",
        else
            cout << "In Special character";
    return 0;
}
```

output :

Input Any character: A  
alphabet

Input Any character: 5  
digit

Input Any character: @

program :

write a program to Input character and if character is in upper case convert into lower case,  
If character is lower case convert into upper case

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

int main ()
{
    char ch;
    clrscr();
    cout<<"Input Any character:",;
    cin>>ch;
    if (ch>=65 && ch<=90)
        ch = ch + 32;
    else
        if (ch>=97 && ch<=122)
            ch = ch - 32;
    cout<<"character is :" << ch;
    return 0;
}
```

output :

Input Any character: A

character is : a

Input Any character: z

character is : Z

Program :

write a program to Input Accno, balance  
Transaction amount, Transaction type and calculate  
Total balance based on the following condition

Transaction type == d

totbal = <balance + tamt

ttype == w, totbal = balance - tamt

```
#include <iostream.h>
```

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

```
int main ()
```

```
{
```

```
int accno;
```

```
char ttype;
```

```
float tamt, bal, tbal;
```

```
clrscr();
```

```
cout<<"\n Input AccNo:";
```

```
cin>> accno;
```

```
cout<<"\n Input balance:";
```

```
cin>> bal;
```

```
tbal = bal;
```

```
cout<<"\n Input Transaction type:";
```

```
cin>> ttype;
```

```
cout<<"\n Input Transaction amt:";
```

```
cin>> tamt;
```

```
if(ttype == 'd')
```

```
    tbal = bal + tamt;
```

```
else
```

```
if(ttype == 'w' && tamt < bal)
```

```
    tbal = bal - tamt;
```

```
else
```

```
cout << "In Invalid Transaction";  
cout << "In Total balance : " << tbal;  
return 0;
```

Output:  
~~~~~

Input AccNo : 12308037

Input balance : 25000

Input Transaction type: d

Input Transaction amt: 5000

Total balance : 30000

Input AccNo : 12308037

Input balance : 25000

Input Transaction type : w

Input Transaction amount : 5000

Total balance : 20000

Input AccNo : 12308037

Input balance : 50000

Input Transaction type : w

Input Transaction amount : 70000

Invalid Transaction

Total balance : 50000

Nested if :-

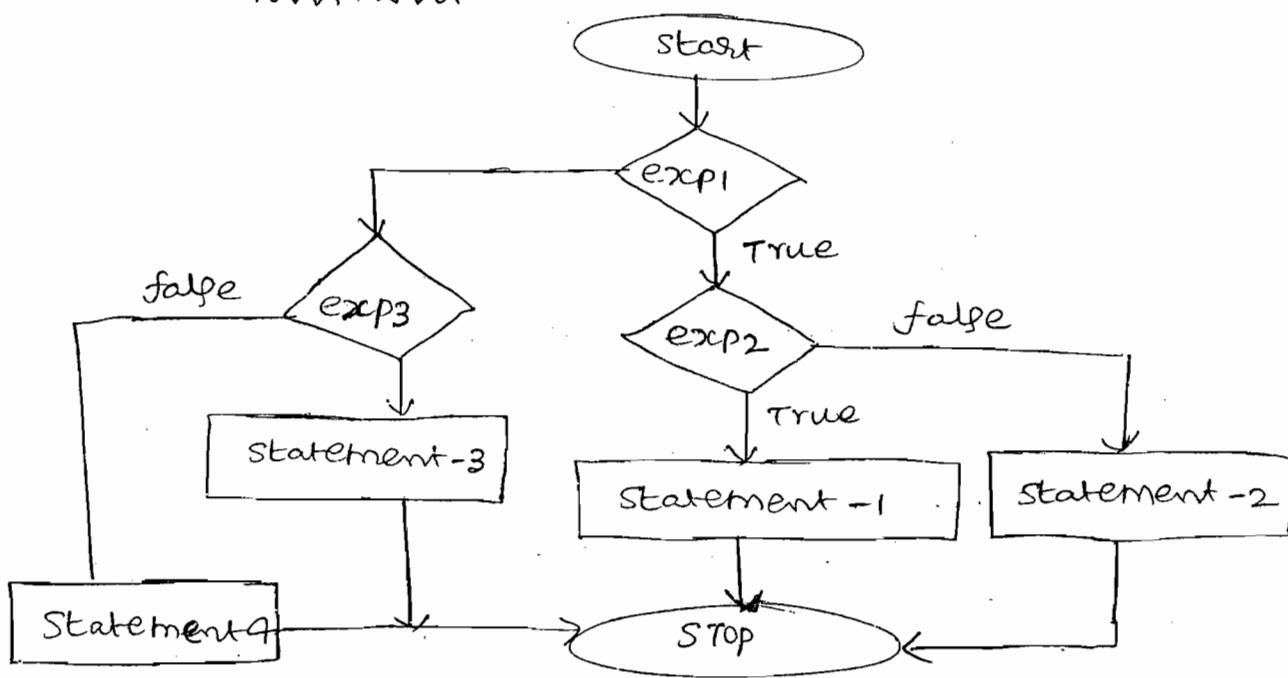
\* Nested means within or immediately.

\* Defining if statement with another if statement  
is called nested.

Syntax :

```
if (expression -1)  
    if (expression -2)  
        Statement -1;  
    else  
        Statement -2;  
    else  
        if (expression -3)  
            Statement -3;  
        else  
            Statement -4;
```

flow chart :-



Program :

Write a program to Input name and three Subject marks Find result without using "AND" or "OR" operators

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

int main()
{
    char name[20];
    int sub1, sub2, sub3;
    clrscr();
    cout << "In Input Name : ";
    cin >> name;
    cout << "In Input 3 subjects : ";
    cin >> sub1 >> sub2 >> sub3;
    if (sub1 >= 40)
        if (sub2 >= 40)
            if (sub3 >= 40)
                cout << "In pass ";
            else
                cout << "In fail ";
        else
            cout << "In fail ";
    else
        cout << "In fail ";
    return 0;
}
```

Output :

Input Name : Balk  
Input 3 subjects :  
80 95 90  
Pass

## Switch Statement :

\* Switch is a Selection Statement or branching Statement.

\* It uses 3 statements

1. switch.

2. case.

3. default.

## switch :

\* Causes control to branch to one of the list of possible statements in the block defined by statement

\* The branched statement-to statement is determined by evaluating Expression It must return an Integral type

## case :

The list of possible branch points with in statements is determined by preceding substatements with case constant Expression

• case <constant expression> :

## Syntax :

Switch (expression)

{

case constant expression:

Statement 1,  
break;

case constant expression:

Statement 2,

.....

case constant expression:

Statement 3;

break;

default :

Statement X;

}

\* These statements are executed based on return value of Expression. If return value expression is equal to any one of these cases It executes the default case

\* case must be terminated with break it will continue the next case

\* Expressions and constant Expression must be of type in integer type

\* case must be followed by constant but not variable

\* The default case can be written anywhere in the switch.

Rules of Switch cases :

~~~~~ ~~~ ~~~~

1. Duplicate cases are not allowed.
2. expression must of type integer.
3. default can be written in anywhere

Program :

write a program to display the following menu

1. Area of The Rectangle,
- 2 Area of The Square
3. Area of The Triangle
- 4 Area of the parallelogram
5. Area of The Rombus
6. Area of The circle

Input your option.

```
#include <iostream.h>
#include <conio.h>
int main()
{
    float base, height, length, breadth, side, area;
    float r, d1, d2;
    int opt;
    clrscr();
    cout<<"In 1. Area of The Rectangle";
    cout<<"In 2. Area of the Square";
    cout<<"In 3. Area of The Triangle";
    cout<<"In 4. Area of the parallelogram";
    cout<<"In 5. Area of The Rombus";
    cout<<"In 6. Area of The circle";
    cout<<"In Input your option:";
    cin>> opt;
    switch(opt)
    {
        case 1:
```

case 1:

```
cout << "In Input length and breadth:";  
cin >> length >> breadth;  
area = length * breadth;  
cout << "In Area of The Rectangle:" << area;  
break;
```

case 2:

```
cout << "In Input side:";  
cin >> side;  
area = side * side;  
cout << "In Area of The Square:" << area;  
break;
```

case 3:

```
cout << "In Input base and height:";  
cin >> base >> height;  
area = 0.5 * base * height;  
cout << "In Area of The Triangle:" << area;  
break;
```

case 4:

```
cout << "In Input base and height:";  
cin >> base >> height;  
area = base * height;  
cout << "In Area of The parallelogram:" << area;  
break;
```

case 5:

```
cout << "In Input d1 and d2:";  
cin >> d1 >> d2;  
area = 0.5 * d1 * d2;
```

case 6:

```
cout << "In Input r:",  
cin >> r;  
area = 3.14 * r * r;  
cout << "In Area of the circle:" << area;  
break;  
default:  
cout << "In Invalid option",  
};  
return 0;
```

OUTPUT :

Input your option: 6

Input r: 7

Area of the circle : 153.86000,

Input your option : 4

Input base and height : 15 12

Area of the parallelogram : 180

Input your option : 7

Invalid option.

---

Program:

~~~~~

write a program to find given character is vowel or not

```
#include <iostream.h>
```

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

```
int main()
```

```
{
```

```
char ch;
```

```
int opt;
```

```
clsscr();
```

```
cout<<"In Input Any character:",
```

```
cin>>ch;
```

```
switch(ch),
```

```
{
```

```
case 'A':
```

```
case 'E':
```

```
case 'I':
```

```
case 'O':
```

```
case 'U':
```

```
cout<<"In vowel",
```

```
break;
```

```
default:
```

```
cout<<"In not vowel",
```

```
}
```

```
return 0;
```

```
}
```

OUTPUT  
~~~~~

Input Any character: o

Vowel

Input Any character: a

Not vowel

## Loops on Iterators

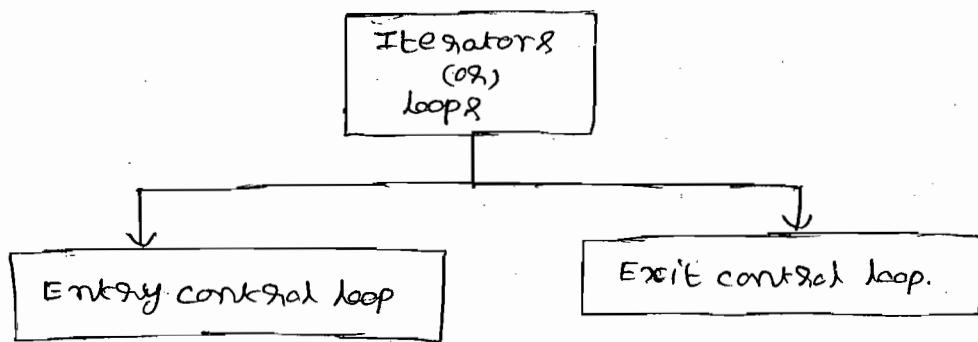
~~~~ ~~~ ~~~

Iterators are used To repeat one or more than one statements no. of times are until given condition.

c++ Language supports two types of loops on iterators.

1. Entry control loop.

2. Exit control loop.



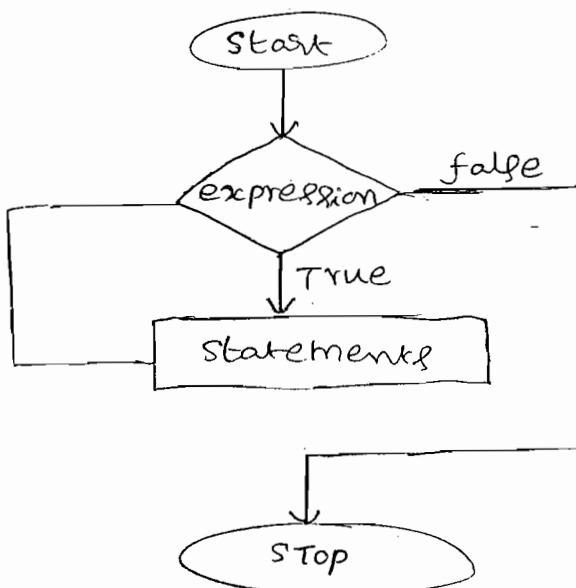
1. Entry control loop :

~~~~ ~~~ ~~~

In Entry control loop condition as

Expression is evaluated before Executing statements.

flow chart :



These are two types of Entry control loops, There are.

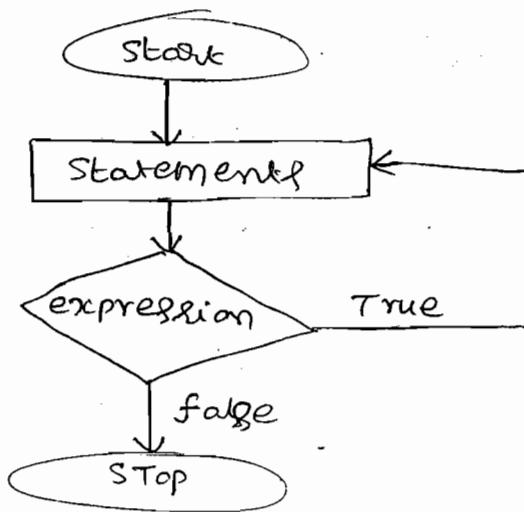
① while loop.

② for loop.

② Exit control loop :

In Exit control loop It executes statements first then Evaluate condition. The statements inside the exit control loop executed at least one time.

flowchart :



\* These are one type of Exit control loop

(i) do-while loop.

\* Every loop required three statements

(i) Initialization .

(ii) condition .

(iii) updation .

(i) Initialization :

a statement which defines initial properties of a condition.

(ii) condition :

a statement which defines how many times the contents of loops to be executed.

### (iii) updation :

a statement which update the condition.

### while loop :

- \* while is a Entry control loop.
- \* It is a function is simply to repeat statements while expression is true

### Syntax :

white (expression)

simple - statement,

(or)

### Syntax :

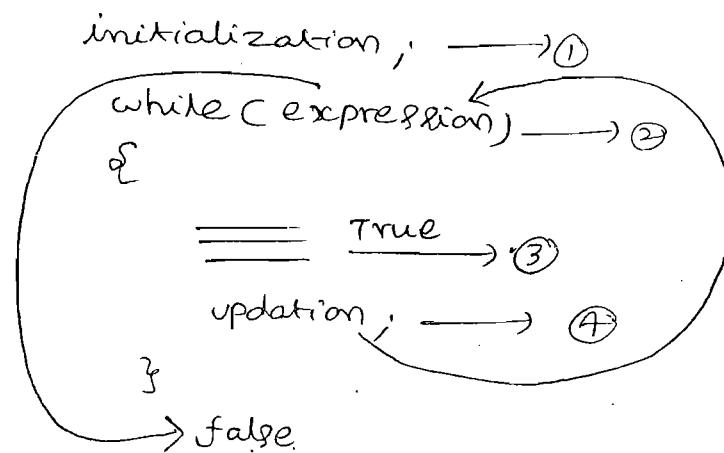
white (expression)

{

compound statement;

}

### Execution of while loop :



program :

write a program to print 1 to 100 numbers.

```
#include <iostream.h>
#include <conio.h>
int main ()
{
    int num=1;
    clrscr();
    while (num<=100)
    {
        cout<<"\n"<<num;
        num=num+1;
    }
    return 0;
}
```

output

1  
2  
3  
4  
5  
1  
1  
1  
100

program:

write a program to print sum of 1 to 100 numbers.

```
#include <iostream.h>
#include <conio.h>
int main ()
{
    int num=1, sum=0;
    clrscr();
    while (num<=100)
    {
        sum = sum+num;
        num = num+1;
    }
}
```

output

sum is : 5050

```
cout<<"\n Sum is :"<<sum;
return 0;
```

Program :

write a program sum of squares of the natural numbers up to 20

$$1^2 + 2^2 + 3^2 + 4^2 + 5^2 + \dots + 20^2$$

```
#include <iostream.h>
```

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

```
int main()
```

```
{
```

```
int num = 1, sum = 0;
```

```
clrscr();
```

```
while (num <= 20)
```

```
{
```

```
sum = sum + (num * num);
```

```
num = num + 1;
```

```
}
```

```
cout << "The sum is :" << sum;
```

```
return 0;
```

```
}
```

Program :

write a program sum of cubes of the Natural Numbers up to 10

$$1^3 + 2^3 + 3^3 + 4^3 + 5^3 + 6^3 + 7^3 + 8^3 + 9^3 + 10^3$$

```
#include <iostream.h>
```

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

```
int main()
```

```
{
```

```
int num = 1, sum = 0;
```

```
clrscr();
```

Output

Sum is : 2870

```

0           while (num <= 10)
0           {
0               sum = sum + (num * num * num)
0               num = num + 1;
0           }
0           cout << "In sum is : " << sum;
0           return 0;
0       }

```

Output :

~~~~~  
sum is : 3025

Program :

write a program sum of squares of even numbers  
up to 20

$$2^2 + 4^2 + 6^2 + 8^2 + 10^2 + 12^2 + 14^2 + 16^2 + 18^2 + 20^2$$

# include <iostream.h>

# include <conio.h>

int main ()

{

int sum=0, num=1;

Output

clscr();

~~~~~  
sum is : 1540

while (num <= 20)

{

if (num % 2 == 0)

sum = sum + (num \* num);

num = num + 1;

}

cout << "In sum is : " << sum;

return 0;

}

Program :

write a program to find given number is Amstrong number or not.

```
#include <iostream.h>
#include <conio.h>
int main ()
{
    int num, r, sum=0, num1;
    clrscr();
    cout<<"\n Input Any Number : ";
    cin>> num;
    num1 = num;
    while (num1 != 0)
    {
        r = sum % 10;
        sum = sum + (r * r * r);
        num1 = num1 / 10;
    }
    if (num1 == sum)
        cout<<"\n Amstrong";
    else
        cout<<"\n Not Amstrong";
    return 0;
}
```

Output : Input Any Number : 153

$$153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$$

Amstrong.

program :

write a program to find a given number is  
palindrome or not

```
#include <iostream.h>
```

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

```
int main()
```

```
{
```

```
int num, r, rev=0, num1;
```

```
clrscr();
```

```
cout<<"\n Input Any Number : ",
```

```
cin>>num;
```

```
num1 = num;
```

```
while (num!=0)
```

```
{
```

```
    r = num % 10;
```

```
    rev = (rev * 10) + r;
```

```
    num = num / 10;
```

```
}
```

```
if (num1 == rev)
```

```
cout<<"\n palindrome";
```

```
else
```

```
cout<<"\n Not palindrome";
```

```
return 0;
```

```
}
```

output :

Input Any Number : 121

palindrome

Input Any Number : 125

Not palindrome.

Program :

write a program to find given number  
if prime or not

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int num, i=1, count=0;
    clrscr();
    cout<< "Input Any Number : ";
    cin>> num;
    while(i <= num)
    {
        if(num % i == 0)
            count++;
        i = i + 1;
    }
    if(count == 2)
        cout<< " prime Number ";
    else
        cout<< " Not prime ";
    return 0;
}
```

Output

Input Any Number: 3

prime Number

Input Any Number: 6

Not prime

Program :

write a program to find factorial of  
given Number.

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int num, fact=1;
```

```

class();
cout<<"\n Input Any Number : ";
cin>> num;
if (num == 0)
    cout<<"\n Factorial is :" << fact;
else
{
    while (num >= 1)
    {
        fact = fact * num;
        num--;
    }
    cout<<"\n Factorial is :" << fact;
}
return 0;
}

```

Output :

Input Any Number : 5

Factorial is : 120

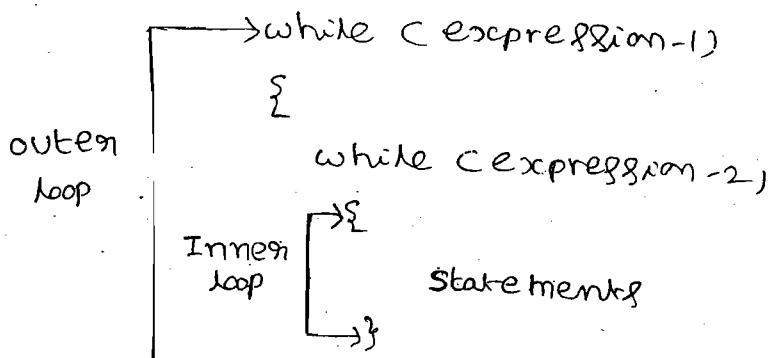
==

Nested While :-

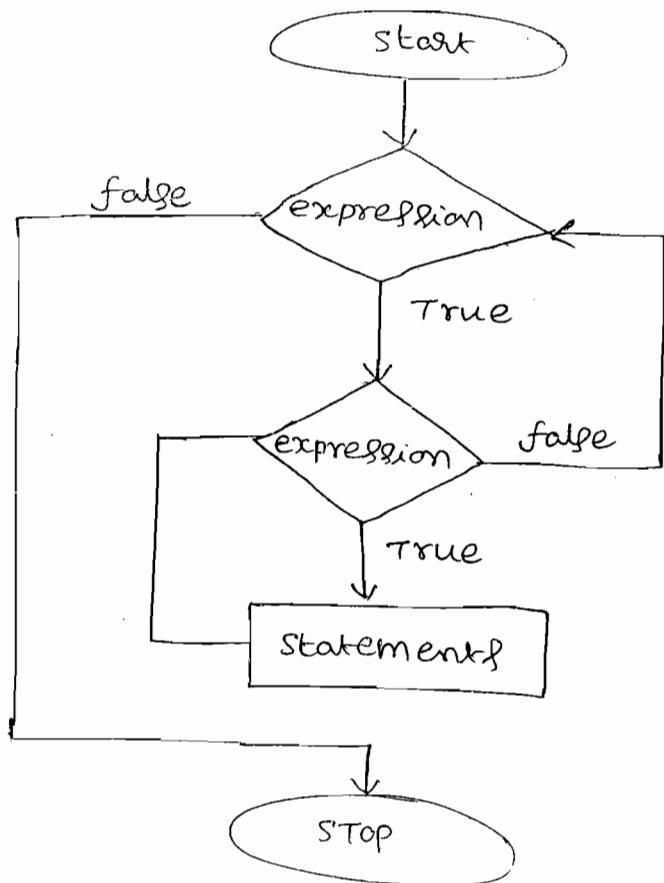
      |  
      |  
      |

\* Defining while loop with in another while  
loop is called Nested while

Syntax :



flow chart :



program :

Write a program to print prime numbers between 2 to 100

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int num=2, i, count;
    clrscr();
    while (num<=100)
    {
        i=1;
        count=0;
        while (i<=num)
        {
            if (num % i == 0)
                count++;
            i++;
        }
        if (count==2)
            cout << num;
        num++;
    }
}
```

output

|    |   |
|----|---|
| 2  | ○ |
| 3  | ○ |
| 5  | ○ |
| 7  | ○ |
| 11 | ○ |
| 13 | ○ |
| 17 | ○ |
| 19 | ○ |
| 23 | ○ |
| 29 | ○ |
| 31 | ○ |
| 37 | ○ |
| 41 | ○ |
| 43 | ○ |
| 47 | ○ |

```

        count++;
      i++;
    }
    if (count == 2)
      cout << "\n" << num;
    num = num + 1;
}
return 0;
}

```

program :

write a program to display the tables up to 20

```

#include <iostream.h>
#include <conio.h>
int main()
{
  int i, j;
  clrscr();
  i = 1;
  while (i <= 20)
  {
    j = 1;
    while (j <= 20)
    {
      cout << "\n" << i * j;
      j = j + 1;
    }
    getch();
    cout << "\n";
    i = i + 1;
  }
  return 0;
}

```

program :

write a program to display the following output

\*  
\* \*  
\* \* \*  
\* \* \* \*  
\* \* \* \* \*  
\* \* \* \* \* \*  
\* \* \* \* \* \* \*  
\* \* \* \* \* \* \* \*  
\* \* \* \* \* \* \* \* \*

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int i, j;
    clrscr();
    i = 1;
    while (i <= 10)
    {
        j = 1;
        while (j <= i)
        {
            cout << " " << "*";
            j = j + 1;
        }
        cout << "\n";
        i = i + 1;
    }
    return 0;
}
```

for loop :-

\* for is an Entry control loop, where condition is evaluated before executing statements.

Syntax :

```
for ([expr1]; [expr2]; [expr3])  
    Statement;
```

Another syntax :

~~~~ ~~~~

```
for ([expr1]; [expr2]; [expr3])  
{  
    Statement-1;  
    Statement-2;  
    :  
    Statement-n;  
}
```

expression-1 :

~~~~ ~~~~

It is called as initialization

Statement This statement is evaluated only once when first time Enter into the loop.

expression-2 :

~~~~ ~~~~

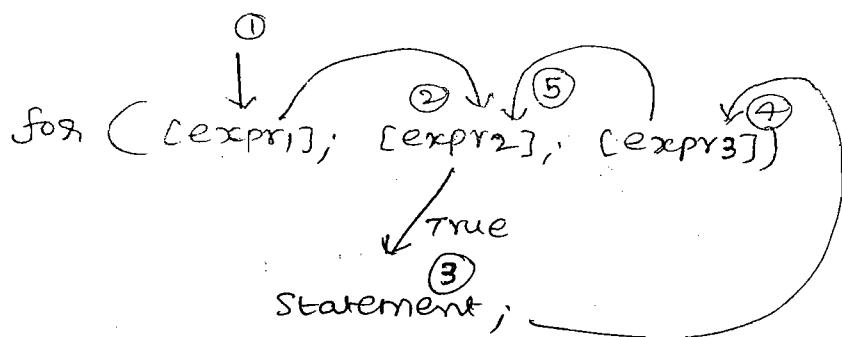
It is called as condition or boolean Expression This Statement define how many times the contents of loop to be executed.

expression-3 :

~~~~ ~~~~

It is called as update statement This statement update the condition.

Execution process of for loop :



other syntaxes :

for( i   expr2, expr3)

    Statement;

for( expr1 ; ; expr3)  
{  
}

for( ; ; expr3)  
{  
}

for( i   expr2 ; )  
{  
}

for( ; ; )  
{  
}

Program :

write a program to display Even Numbers  
1 to 100 numbers

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

int main()
{
    int num;
    clrscr();
    for (num = 1; num <= 100; num++)
    {
        if (num % 2 == 0)
            cout << "\n" << num;
    }
    return 0;
}
```

Program :

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

int main()
{
    int i = 0;
    clrscr();
    for (i = 0; i < 20; i++)
    {
        switch (i)
        {
            case 0 : i += 5;
            case 1 : i += 2;
            case 5 : i += 5;
        }
    }
}
```

```
cout << "1t" << i;  
}  
return 0;  
}
```

OUTPUT

16 21

Nested for loop :  
~~~~~ ~~~~

\* defining for loop with in for loop is called  
Nested for loop.

Syntax :  
~~~~~

```
for(expr1; expr2; expr3)  
{
```

```
    for(expr1; expr2; expr3)
```

{

    statements;

}

    statements;

}

program :  
~~~~~

write a program to find palindrum  
numbers between 1 to 100

11 22 33 44 55 66 77 88 99

```
#include <iostream.h>
```

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

```
int main(),
```

{

```
    int num, rev, num1, r;
```

```
    clrscr();
```

```

for (num=1; num<100; num++)
{
    for (num1=num, rev=0, num1!=0; num1=num1/10)
    {
        r = num1 % 10;
        rev = (rev*10) + r;
    }
    if (rev == num)
        cout << num;
}
return 0;
}

```

program :

write a program to display the following  
out put

```

*****  

****  

***  

**  

*
```

```

#include <iostream.h>  

#include <conio.h>  

int main()  

{
    int i, j;  

    clrscr();  

    for (i=1; i<=5; i++)  

    {
        for (j=1; j<=5; j++)
    }
}

```

```
    cout << "\n";  
}  
return 0;  
}
```

Program:

~~~~~

Write a program to display the following output

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5  
1 2 3 4 5 6  
1 2 3 4 5 6 7  
1 2 3 4 5 6 7 8  
1 2 3 4 5 6 7 8 9  
1 2 3 4 5 6 7 8 9 10
```

```
#include <iostream.h>  
#include <conio.h>  
int main()  
{  
    int i, j, k;  
    clrscr();  
    for (i=0; i<=10; i++)  
    {  
        for (j=1, k=1; j<=i; j++, k++)  
        {  
            cout << " " << k;  
        }  
        cout << "\n";  
    }  
}
```

Program :

write a program to display the following  
output :

a  
a b  
a b c  
a b c d  
a b c d e

```
#include <iostream.h>
#include <conio.h>
int main()
{
    char i, j;
    clrscr();
    for (i = 97; i <= 101; i++)
    {
        for (j = 97; j <= i; j++)
        {
            cout << " " << j;
        }
        cout << "\n";
    }
    return 0;
}
```

---

do-while loop :

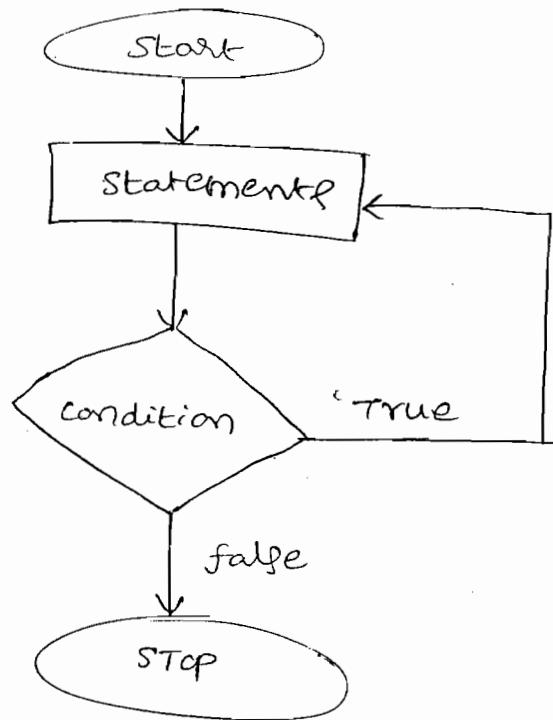
\* do-while is called exit control loop, where statements are executed first and evaluate the condition.

Syntax :

```
do
{
    statements,
}
while (condition);
```

\* In do-while the contents of loop is executed at least one time.

flow chart :



Program :

```
#include <iostream.h>
#include <conio.h>
int main()
{
    float l, b, r, s, h;
    float area, circumference, perimeter, volume;
    int opt;
    clrscr();
    do
    {
        cout << "In 1. Area of the Rectangle";
        cout << "In 2. Perimeter of the Rectangle";
        cout << "In 3. Area of the Square";
        cout << "In 4. Perimeter of the Square";
        cout << "In 5. Area of the Triangle";
        cout << "In 6. Area of the Circle";
        cout << "In 7. Circumference of the Circle";
        cout << "In 8. Area of the Parallelogram";
        cout << "In 9. Area of the Sphere";
        cout << "In 10. Volume of the Sphere";
        cout << "In Input your option : ";
        cin >> opt;
        switch(opt)
        {
            case 1:
                cout << "In Input l and b values : ";
                cin >> l >> b;
                Area = l * b;
                cout << "In Area of the Rectangle : " << Area;
                break;
        }
    } while (opt != 10);
}
```

case 2:

```
cout << "In Input l and b values:";  
cin >> l >> b;  
perimeter = 2 * (l + b);  
cout << "In perimeter of the Rectangle:"  
break;
```

perimeter;

case 3:

```
cout << "In Input s value:";  
cin >> s;  
area = s * s;  
cout << "In Area of the Square:" << area;  
break;
```

case 4:

```
cout << "In Input s value:";  
cin >> s;  
perimeter = 4 * s;  
cout << "In perimeter of the Square:" << perimeter;  
break;
```

case 5:

```
cout << "In Input b and h values:";  
cin >> b >> h;  
area = 0.5 * b * h;
```

```
cout << "In Area of the Triangle:" << area;  
break;
```

case 6:

```
cout << "In Input r value:";  
cin >> r;  
area = 3.14 * r * r;  
cout << "In Area of the circle:" << area;  
break;
```

case 7:

```
cout << "In Input r value:";  
cin >> r;  
circumference = 2 * 3.14 * r;  
cout << "In circumference of the circle" << circumfer-  
-ence;  
break;
```

case 8:

```
cout << "In Input b and h values:";  
cin >> b >> h;  
area = b * h;  
cout << "In Area of the parallelogram:" << area;  
break;
```

case 9:

```
cout << "In Input r value:";  
cin >> r;  
area = 4 * 3.14 * r * r;  
cout << "In Area of the Sphere:" << area;  
break;
```

case 10:

```
cout << "In Input r value:";  
cin >> r;  
volume = 4/3 * 3.14 * r * r * r;  
cout << "In Volume of the Sphere:" << volume;  
break;
```

}

```
} while(opt<=10);  
return 0;
```

y

break :-

\* break is a keyword.

\* It passes the control.

Syntax : break;

\* The break statement causes control to pass to the statement following the innermost enclosing while, do, for or switch statement.

Program :

```
#include <iostream.h>
```

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

```
int main()
```

```
{
```

```
    int num = 2,
```

```
        while (1)
```

```
{
```

```
    cout << num;
```

```
    num = num + 2;
```

```
    if (num > 50)
```

```
        break;
```

```
}
```

```
    return 0;
```

```
}
```

---

continue :

\* It is a keyword and passes the control.

\* causes control to pass to the end of the innermost enclosing while, do or for statement, at which point the loop continuation condition is re-evaluated.

Syntax : continue;

Program : write a program to display natural numbers

```
#include <iostream.h>
#include <conio.h> *
int main()
{
    int num;
    clrscr();
    for (num=1; num<=500; num++)
    {
        cout<<"\t" << num;
        continue;
    }
    return 0;
}
```

Program :

write a program to display the odd numbers up to 100

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int i;
    clrscr();
```

```
for( i=1; i<=100; i++)
{
    if (i%2 != 0)
        cout << "n" << i;
    continue;
}
return 0;
```

goto :-

\* goto is a keyword.

\* It Transfers control.

Syntax :

```
goto <identifier>;
```

\* control is unconditionally transferred to the location of a local label specified by <identifier>

program :

write a program to display the following output

```
***** * * * * *
* *
* HELLO BALANAGENDRUDU *
* *
* *
* *
* * * * * * * *
```

```

#include <iostream.h>
#include <conio.h>
int main ()
{
    int r, c;
    clrscr();
    for (r=5; r<=15; r++)
    {
        gotoxy (5, r);
        cout<< "*";
        gotoxy (10, r);
        cout<< "*";
    }
    for (c=5; c<=40; c++)
    {
        gotoxy (c, 5);
        cout<< "*";
        gotoxy (c, 15);
        cout<< "*";
    }
    gotoxy (10, 8);
    cout<< "HELLO BALANAGENDRUDU";
    return 0;
}

```

Program:

Write a program to display the following output

```

===== ===== ===== =====
||| ||| ||| |||
||| WELCOME TO C++
||| ||| ||| |||
===== ===== ===== =====

```

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

int main()
{
    int r, c;
    clrscr();
    for (r=5; r<=15; r++)
    {
        gotoxy (5, r);
        cout << "=";
        gotoxy (50, r);
        cout << "=";
    }
    for (c=5; c<=50; c++)
    {
        gotoxy (c, 5);
        cout << "=";
        gotoxy (c, 15);
        cout << "=";
    }
    gotoxy (10, 8);
    cout << "* WELCOME TO C++ *";
    return 0;
}
```

---

```
cout << setiosflags(ios::hex) << x << endl(3)
```

```
return 0;
```

```
}
```

8-bit	— 1 byte
1024 bytes	— 1 KB
1024 KB	— 1 MB
1024 MB	— 1 GB
1024 GB	— 1 TB

Pointers :-

- \* Pointer is derived data type.

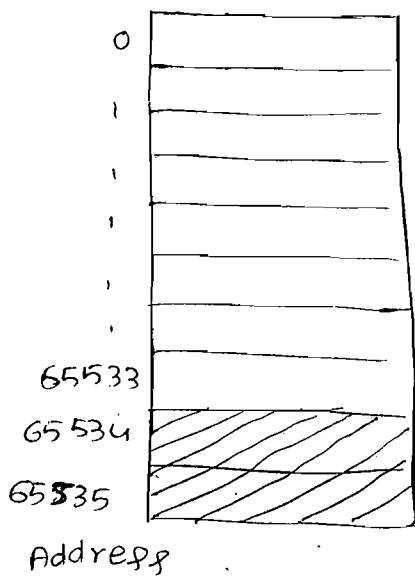
- \* A variable declared using pointer datatype is called pointer variable.

- \* Pointer variable hold An address of memory location or value.

Q what is Address?

A → It is an unsigned integer value given to each location in memory

→ memory is a collection of bytes each byte in memory operating system assign one unique id called address.



Q How to print an address of variable?

- A → It is printed used ampersand (&) operator  
It means address of operator

cout << &a; → 65534

cout << &b; → 65532

→ ampersand to get the address

Ex:- int x=10;

int y;

y=&x; → Error.

y is a value type variable cannot hold address.

Syntax of pointers :-

datatype \*variablename;  
                    ↓  
                    pointer type

→ '\*' indicated pointer type.

→ '\*' indicated value at address.

→ '\*' indicated dereferencing operator.

→ '\*' indicated point to address.

Ex:- int \*a;

a=10; → Error

Advantages :-

1. Dynamic memory allocation (DMA).

2. It increases efficiency of the program (avoiding wastage of memory).

3. Helpful building complex data types like ~~array~~ and dynamic data structures like linked list.
4. Sharing data between the functions (call by address).
5. System programming.

Disadvantage of pointers :-

1. pointers are not secure.

\* The size of the pointers varies from one compiler to another. 16-bit compilers allocates 2-bytes for pointers.  
32-bit compilers allocates 4-bytes for pointers.

Ex:-      int \*p; → 2 bytes.  
               float \*q; → 2 bytes.  
               char \*r; → 2 bytes.  
               double \*d; → 2 bytes.

Program:

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

int main()
{
    int *p;
    char *q;
    float *r;
    clrscr();
    cout << " \n size of integer pointer " << sizeof(p);
    cout << " \n size of character pointer " << sizeof(q);
    cout << " \n size of float pointer " << sizeof(r);
    return 0;
}
```

O/P  
~~~~~  
2-bytes  
2-bytes  
2-bytes

→ \*y nothing, but value at

int main()

{

int a=10;

int \*p;

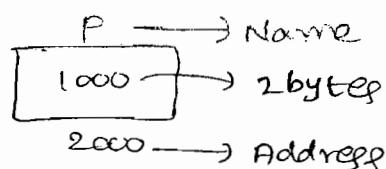
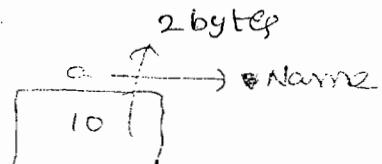
p = &a;

cout << p; → 1000

cout << \*p → 10

↓  
value at 1000

}



[ int x=10;  
int y;  
y=&x; → Error ]

→ Each pointer variable is declared with an appropriate data type. The data type defines the type of value referred by pointers.

② Find output?

#include <iostream.h>

#include <conio.h>

int main()

{

int x=10;

float \*y;

y = &x

cout << \*y;

return 0;

}

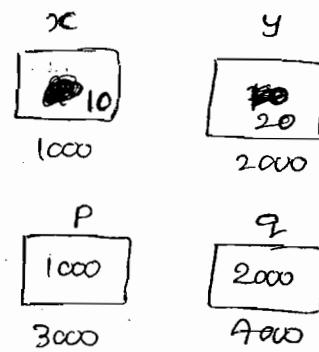
The above program displays compile time error because the float pointer cannot hold addresses or integer values.

Program :-

```

#include <iostream.h>
#include <conio.h>
int main()
{
    int x, y;
    int *P, *Q;
    clrscr();
    cout << "Input x, y values";
    cin >> x > y;
    P = &x;
    Q = &y;
    *P = *P + *Q;
    *Q = *P - *Q;
    *P = *P - *Q;
    cout << "After Swapping\n";
    cout << "x = " << x;
    cout << "y = " << y;
    return 0;
}

```



$$*P = *P + *Q$$

$$\begin{aligned} \text{Value at } 1000 &= \text{Value at } 1000 \\ &\quad + \text{Value at } 2000 \\ &= 10 + 20 = 30 \end{aligned}$$

$$*Q = *P - *Q$$

$$\begin{aligned} \text{Value at } 2000 &= \text{Value at } 1000 - \text{Value at } 2000 \\ &= 30 - 20 = 10 \end{aligned}$$

~~$$*P = *P - *Q$$~~

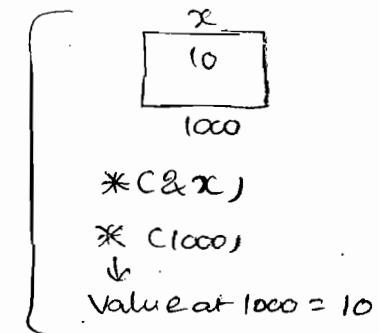
$$\begin{aligned} \text{Value at } 1000 &= \text{Value at } 1000 \\ &\quad - \text{Value at } 2000 \\ &= 30 - 10 = 20 \end{aligned}$$

Program :-

```

#include <iostream.h>
#include <conio.h>
int main()
{
    int x = 10;
    clrscr();
    cout << "x = " << *(&x);
    return 0;
}

```



$$*(\&x)$$

$$*(1000)$$

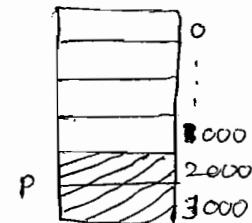
$$\downarrow$$

$$\text{Value at } 1000 = 10$$

② what is Dangling pointer ?

- ③ → A pointer variable which hold an address of unreference memory location is called dangling pointer.
- These pointers are not secured.
- This leads to memory leakage.

```
int main()
{
    int *p; → 2bytes
}
```



→ These dangling pointers are eliminated to programming language like Java and .Net

Void pointers :-

- \* void is an empty datatype.
- \* pointers can also be declared of void.
- \* void pointers can not be dereferenced without explicit casting. This is because the compiler cannot determine the size of the variable or object at the pointer points to.
- \* a pointer variable which holds an address of any type of value.

Syntax:-

```
void *variable-name;
```

Program:-

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int x=10;
```

```

float y=1.5;
void *p;
p=&x;
cout<<"\nx="<<*(int*)p;
p=&y;
cout<<"\ny="<<*(float*)p;
return 0;
}

```

O/P  
 10  
 1.5

constant pointer :-

When the constant modifier is used with pointer variable that variable can not modified the value or variable that the pointer points to.

Syntax :-

const datatype \*variable-name;

\* This pointer is helpful in functions to restrict not changing the value of actual argument.

Program :-

```

#include <iostream.h>
#include <conio.h>
int main()
{
  int x=10;
  const int *p;
  p=&x;
  clrscr();
  cout<<"x="<<*p;
  // p=x;
  return 0;
}

```

\* The above program displays compile time error because constant pointer can not change the value to  $\infty$ .

Double pointer :-

\* a pointer variable, which hold an address of single pointer variable is called double pointer.

Syntax :-

datatype \*\*variable-name;

Eg :-

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int a=10;
    int *p;
    int **q;
    clrscr();
    p=&a;
    q=&p;
    cout<<"\na = "<<a;
    cout<<"\na = "<<*p;
    cout<<"\na = "<<**q;
    return 0;
}
```

$*p$

value at 1000 = 10

$**q$

Value at (Value at 2000)

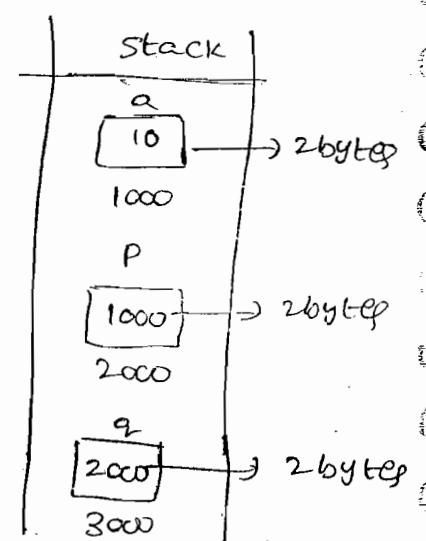
Value at (1000) = 10

O/P

a=10

a=10

a=10

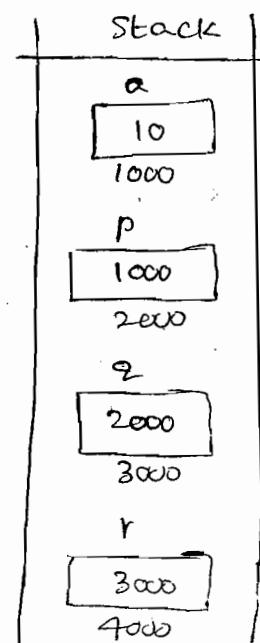


Program:

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int a=10;
    int *p, **q, ***r;
    clrscr();
    p=&a;
    q=&p;
    r=&q;
    cout<<"\na="<<a;
    cout<<"\na="<<*p;
    cout<<"\na="<<**q;
    cout<<"\na="<<***r;
    return 0;
}
```

O/P

a=10  
a=10  
a=10  
a=10



\*p  
↳ Value at 1000 = 10

\*\*q  
↳ Value at 2000 = 1000

Value at 1000 = 10

\*\*\*r  
↳ Value at 3000 = 2000

Value at 2000 = 1000

Value at 1000 = 10

Output

a=10  
a=10  
a=10  
a=10

## Arrays

- \* Array is a derived data type.
- \* Array is a collection similar type elements or values.
- \* Array is an implicit pointer.

Advantage of using Arrays:-

- \* Avoiding declarations of no. of variables.
- \* All the values referred by using one name.
- \* Grouping or allocating memory at one place.
- \* Efficient in reading and writing data.

disadvantages:-

- \* wastage of memory because of static behaviour  
(The size of the Array not increase or decrease)

Types of Arrays:-

They are two types of arrays.

1. Single dimensional Array.
2. multi dimensional Array.

Single dimensional Array :-

- \* An array whose elements are referred using one subscript.

- \* Single dimensional array is implicit single pointer.

Syntax:-

datatype array-name [size];

- \* Size must be constant.

Ex :- int n=10;

int a[n]; → Invalid

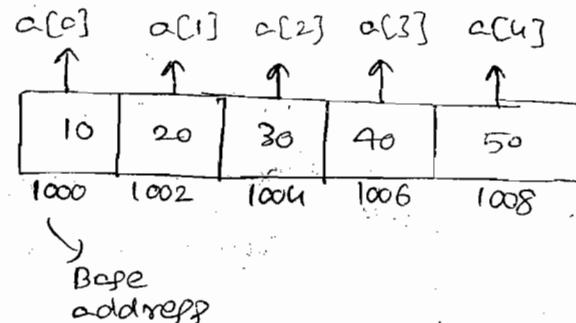
int b[5]; → Valid.

`int c[-3];` → invalid

\* Size is always positive.

Eg:-

`int a[5];`  
↓  
array name  
pointer name



$\ast (\text{base address} + \text{offset} * \text{size of datatype})$

$$a[0] = 10 \rightarrow \ast(a + 0 * 2) \rightarrow \ast(1000 + 0) \rightarrow \ast 1000$$

$$a[1] = 20 \rightarrow \ast(a + 1 * 2) \rightarrow \ast(1000 + 2) \rightarrow \ast 1002$$

\* The formula to Refer each location in array is

$\ast (\text{base address} + \text{offset} * \text{size of datatype})$

Eg:-

$$a[2] = 30$$

$$\rightarrow \ast(a + 2 * 2) \rightarrow \ast(a + 4) \rightarrow \ast(1000 + 4) \rightarrow \ast 1004 = 30$$

Comments:-

- Each location in array identify a unique number is called index numbers.
- Index number is logical numbers.

Program:-

write a program to read name and 3 subject marks.

```
#include <iostream.h>
```

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

```
int main ()
```

```
{
```

```
char name [10],
```

```

int subject[3], index;
clsScr();
cout << "In input name";
cin >> name;
cout << "Input 3 subjects";
for (index = 0; index < 3; index++)
    cin >> subject[index]; // *(&subject + index)
cout << "In name " << name;
for (index = 0; index < 3; index++)
    cout << "In " << subject[index];
return 0;
}

```

- \* Array is implicit pointer
- \* implicit means compiler declaring pointer, but not by user  
That is what implicit pointer.

### Program:

To read n elements and printing

```

#include <iostream.h>
#include <conio.h>
int main()
{
    int a[10], n;
    int index;
    clsScr();
    cout << "In input how many elements? ";
    cin >> n;
    cout << "Input elements";
    for (index = 0; index < n; index++)
        cin >> a[index];
}

```

```

cout << "In elements are \n";
for (index=0; index<n; index++)
    cout << "\n" << *(a+index);
return 0;
}

```

\* There is no bound checking on arrays in C & C++. Because arrays are implicit pointers and pointer can hold ~~reserved~~ memory location / unreserved memory location.

```

int a[5] = [10, 20, 30, 40, 50] → initialization
for (i=0; i<10; i++)
    cout << a[i]; → 10, 20, 30, 40, 50, 60, 70
⇒ int a[5]           ↑ value
          |           ↓
          single pointer      int *a[5]
                                ↓
                                double pointer

```

Array of Pointers :-

\* It is an array which holds address of one / more than one value of similar type.

\* It is a double pointer.

\* It is a collection of address

Syntax :-

datatype \*array-name [size];

Ex :- int \*a[5]

Program :-

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

int main()
{
    int x=10, y=20, z=30;
    a[0] = &x;
    a[1] = &y;
    a[2] = &z;
```

cout << \*a[0]; → value at 1000 = 10

cout << \*a[1]; → value at 2000 = 20

cout << \*a[2]; → value at 3000 = 30

return 0;

}

| x  | y  | z  |
|----|----|----|
| 10 | 20 | 30 |

$$\begin{aligned} *a[0] &= *(*(&a[0])) \\ &= *C* \end{aligned}$$

Double dimensional array :-

    mm    mm    mm

\* An array of single dimensional array is called double dimensional array.

\* In double dimensional array data is originally logically dividing into rows and columns.

int subject [3], → int subject [5][3];
 ↑ rows
 ↓ column
 int sum [11]; → int sum [5][11];

\* It is a implicit double pointer.

Syntax:-

datatype array name [row size][col size];

Program :  
~~~~~

write a program to read 3x3 matrix display.

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
int main()
{
    int a[3][3];
    int i, j;
    clrscr();
    cout << "In Input elements of matrix";
    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            cin >> a[i][j];
    cout << "Elements of matrix\n";
    for (i=0; i<3; i++)
    {
        for (j=0; j<3; j++)
            cout << *(*(&a[i])+j);
        cout << "\n";
    }
    return 0;
}
```

Answer :-

O/P

input elements of matrix

1 2 3  
4 5 6  
7 8 9

Elements of matrix

1 2 3  
4 5 6  
7 8 9

Program :  
~~~~~

write a program To read n students of 3 subject  
marks display?

```

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

int main()
{
    int student[10][3];
    int i, j;
    int n;
    clrscr();
    cout << "In input how many subjects ? ";
    cin >> n;
    cout << "In Input marks \n";
    for (i=0; i<n; i++)
        for (j=0; j<3; j++)
            cin >> student[i][j];
    cout << "In marks are \n";
    for (i=0; i<n; i++)
    {
        for (j=0; j<3; j++)
            cout << student[i][j] << " ";
        cout << "\n";
    }
    return 0;
}

```

o/p

Input how many students?

2

marks are

|    |    |    |
|----|----|----|
| 60 | 70 | 80 |
| 80 | 60 | 70 |

Program :

write a program to add two matrices

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int a[3][3], b[3][3], i, j,
        clrscr();
    cout << "Input a matrix: " << endl;
    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            cin >> a[i][j];
    cout << "Input b matrix: " << endl;
    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            cin >> b[i][j];
    cout << " Adding of two matrices: " << endl;
    for (i=0; i<3; i++)
    {
        for (j=0; j<3; j++)
            cout << " " << a[i][j] + b[i][j];
        cout << endl;
    }
    return 0;
}
```

---

Program :

Write a program to multiply the two matrices

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

int main()
{
    int A[3][3], B[3][3], C[3][3], i, j, k;

    clrscr();
    cout<<"\n Input Elements of A matrix: "<<"\n";
    for (i=0; i<3; i++)
    {
        for (j=0; j<3; j++)
        {
            cin>> A[i][j];
        }
    }

    cout<<"\n Input Elements of B matrix: "<<"\n";
    for (i=0; i<3; i++)
    {
        for (j=0; j<3; j++)
        {
            cin>> B[i][j];
        }
    }

    cout<<"\n matrix multiplication: "<<"\n";
    for (i=0; i<3; i++)
    {
        for (j=0; j<3; j++)
        {
            C[i][j]=0;
            for (k=0; k<3; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

```

    C[i][j] = C[i][j] + A[i][k] * B[k][j];
}
cout << " " << C[i][j];
}
cout << "\n";
}
return 0;
}

```

OUTPUT :

~~~~~

Input Elements of A matrix:

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

Input Elements of B matrix:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Matrix multiplication:

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

\_\_\_\_\_

Program :

write a program to find the trace of a matrix;

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int A[3][3], i, j, trace=0;
    clrscr();
    cout<<"\n Input for matrix A:"<<"\n";
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
    {
        cin>>A[i][j];
        if(i==j)
            trace+=A[i][j];
    }
    cout<<"\n Trace of matrix : "<<trace;
    cout<<"\n";
    return 0;
}
```

Output :

Input for matrix A:

```
1 2 3
4 5 6
7 8 9
```

Trace of matrix : 15

program :

write a program to transpose of a matrix:

```
#include <iostream.h>
#include <conio.h>
int main ()
{
    int A[3][3], i, j;
    clrscr();
    cout << "Input for matrix A: " << endl;
    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            cin >> A[i][j];
    cout << endl << "Transpose of matrix A: " << endl;
    for (i=0; i<3; i++)
    {
        for (j=0; j<3; j++)
            cout << " " << A[j][i];
        cout << endl;
    }
    return 0;
}
```

output :

Input for matrix A:

```
1 2 3
4 5 6
7 8 9
```

Transpose of matrix A:

```
1 4 7
2 5 8
3 6 9
```

O

Θ

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

## Dynamic memory allocation (DMA) :-

in notes (3)

- \* "Dynamic" term refers to runtime.
  - \* Allocating or releasing memory during the execution of program is called dynamic memory allocation.
  - \* Allocating memory at runtime is called dynamic memory allocation.
  - \* It allocates the values in heap area.
- Advantages :-
- \* Avoiding wastage of memory.
  - \* Efficiency of program is increased.
  - \* Dynamic memory allocation is achieved with C++ using the following operators.

1. new.

2. delete.

1. new :-

Syntax :-

$\langle \text{pointer-to-name} \rangle = \text{new} \langle \text{name} \rangle [\langle \text{name-initializer} \rangle]$   
- zero >.

The new operator tries to create an object or variable name by allocating size of ( $\langle \text{name} \rangle$ ) bytes in the heap.

The storage duration of the new object or variable is from the point of creation until the operator "delete" deallocated its memory, or until the end of the program.

	int	float	char
Initial values	0	0.0	non printable value
Storage	Heap area		
Lifetime	until delete, until execution of the program		

Q) what is difference between new and malloc?

new  
~~~~~

1. It is a operator.
2. It allocates memory for one or more than one value.
3. It does not required any type casting.

malloc  
~~~~~

1. It is a function.
2. It allocates memory for only one value.
3. Return type of this function is void pointer, it required type casting.

2. delete :-  
~~~~~

Syntax:-  
~~~~~

delete <pointer-to-name>;

The "delete" operator destroys the object or variable <name> by deallocating size of (<name>) bytes pointed to by <pointer-to-name>.)

Program:  
~~~~~

```
#include <iostream.h>
#include <conio.h>
int main()
{
    clrscr();
    new int;
    new float;
    new char;
    return 0;
}
```

Heap area

int

0

float

0.0

char

nu

- \* The new operator reserve memory and return address of that memory location.
- \* Address of dynamically allocated variable hold by a pointer variable.

Ex :-

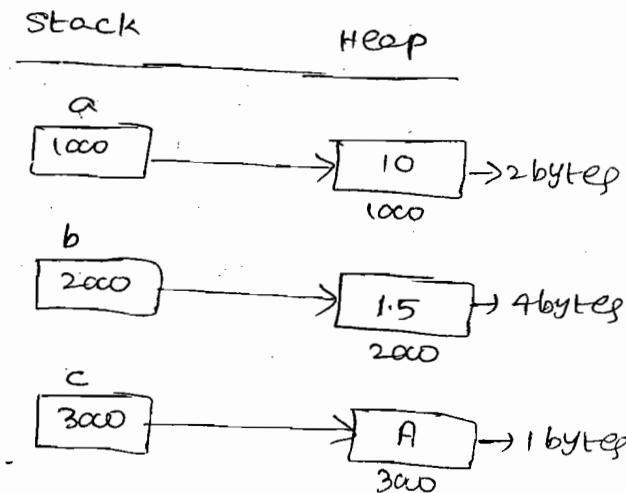
```

int main ()
{
    int *a;
    float *b;
    char *c;

    a = new int;
    b = new float;
    c = new char;

    *a = 10;
    *b = 1.5;
    *c = 'A';

    cout << *a; → 10
    cout << *b; → 1.5
    cout << *c; → A
    return 0;
}
  
```

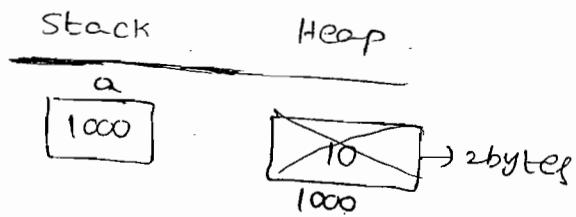


Eg :-

```

int main ()
{
    int *a;
    a = new int;
    *a = 10;

    cout << *a; → 10
    delete a;
    return 0;
}
  
```



\* deleting before execution of the function.

Dynamic Array :- program:

\* write a program to read n values and print

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int *a;
    int n, index;
    clrscr();
    cout << "In input the value of n";
    cin >> n;
    a = new int[n];
    cout << "In input elements of array";
    for(index=0; index<n; index++)
        cin >> a[index];
    cout << "Elements are ";
    for(index=0; index<n; index++)
        cout << " " << a[index];
    return 0;
}
```

Program:

\* write a program to read name and 'n' subject marks?

```
#include <iostream.h>
#include <conio.h>
int main()
{
    char name[10];
    int *Subject, n;
```

```

    clrscr();
    cout << "Input name";
    cin >> name;
    cout << "Input how many subjects?";
    cin >> n;
    Subject = new int[n];
    cout << "Input subjects";
    for (int index=0; index<n; index++)
        cin >> Subject[index];
    cout << "Name" << name;
    cout << "Marks are \n";
    for (index=0; index<n; index++)
        cout << " " << Subject[index];
    return 0;
}

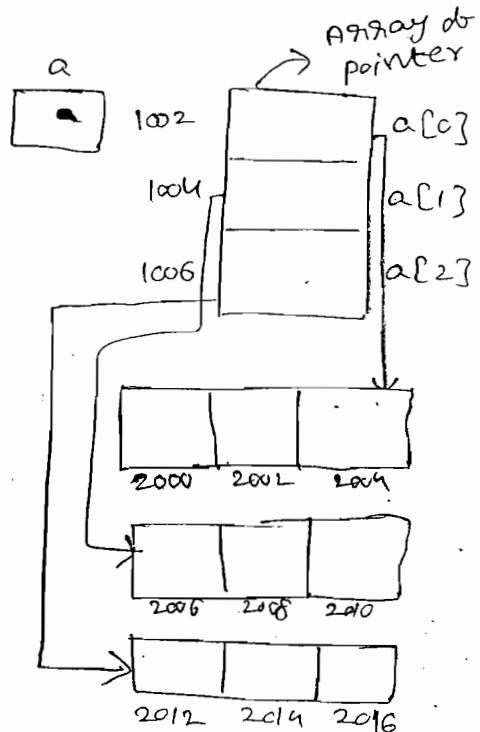
```

creating dynamic double dimensional array :-

1. Declare a double pointer.
2. Create an array of pointers.

3. Create an array of values which hold by each pointer.

Ex :-      int \*\*\*a;  
                 a = new int\*[3];  
                 a[0] = new int[3];  
                 a[1] = new int[3];  
                 a[2] = new int[3];



Program :-

Write a program to create mxn matrix and read elements and display

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int **a;
    int m, n, i, j;
    clrscr();
    cout << "In input how many rows";
    cin >> m;
    cout << "In input how many columns";
    cin >> n;
    a = new int*[m];
    for (int i=0; i<m; i++)
        a[i] = new int[n];
    cout << "In input elements of matrix";
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            cin >> *(*(a+i)+j);
    cout << "In Elements are In";
    for (i=0; i<m; i++)
    {
        for (j=0; j<n; j++)
            cout << " " << *(*(a+i)+j);
        cout << "\n";
    }
    return 0;
}
```

O/P

## Reference Variables :-

- \* Reference is a data type/derived datatype.
- \* Reference variables are alias variables/link variables.
- \* Reference variables are implicit pointer constants.
- \* These are Read only pointers.

## Advantages :-

1. Secured compared with pointers.
2. Sharing data between functions.

## Syntax :-

datatype &Variable-name = Variable;

→ This means Reference

## Note :-

Reference variable must be initialized by assigning another variable.

Ex:-      int b = 20;  
                int &a = b;  
                cout << a; → 20

int &x = 10 (constant)  
                  ↓ invalid.

int x = 10;  
int &y = x;  
int z = 20;  
&y = z; → invalid.

## Program :-

```
#include <iostream.h>
#include <conio.h>
int main()
```

```
int x = 10;  
int &y = x;  
int &z = y;  
char ch();  
cout << "\n" << x;  
cout << "\n" << y;  
cout << "\n" << z;
```

O/P  
10  
10  
10  
60  
60  
60

```
z = 60;  
cout << "\n" << x;  
cout << "\n" << y;  
cout << "\n" << z;  
return 0;
```

}

—

# Functions

• • • • • • • •

Q @ what is Function?

Q A \* A function is self contained block, which contain  
Q set of instructions

Q \* A function is a small program within program.

Q \* A function is a module.

Q \* A function is a named block.

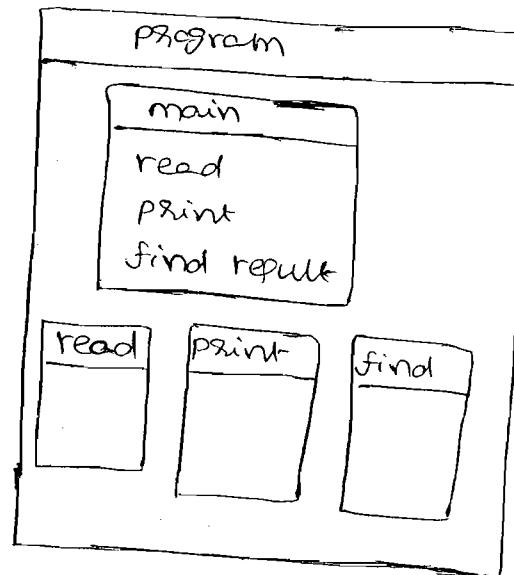
Q \* A function is a reusable code component.

Q \* A function is subroutine.

Q Advantages of functions :-  
Q • • • • •

Q 1. modularity :-  
Q • • •

Q Dividing programming instructions  
Q according to its operations into small modules.



Q 2. Reusability :-  
Q • • •

Q write code once and use many  
Q times.

### 3. Readability :-

Easy to understand application operations.

### 4. Efficiency :-

It increases an Efficiency of the program.

Properties or components of the Functions :-

1. return type

2. function name.

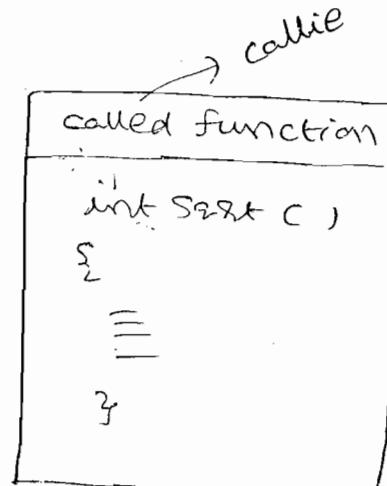
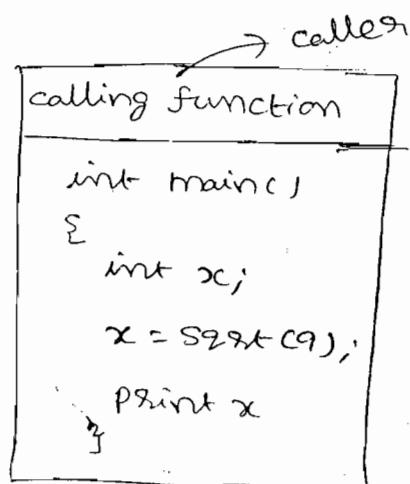
3. parameters.

4. function call.

1. return type :-

→ It is a datatype, which define the type of value return by function.

→ If function does not return any value, its return type is void.



2. function name :-

\* function name must be valid identifier.  
It should not be keyword.

\* function name is user defined name.

### ③ Parameters :-

\* Parameters are local variables, which receive data/values from calling functions.

\* There are two types.

1. Actual parameters/arguments.

2. formal parameters/arguments.

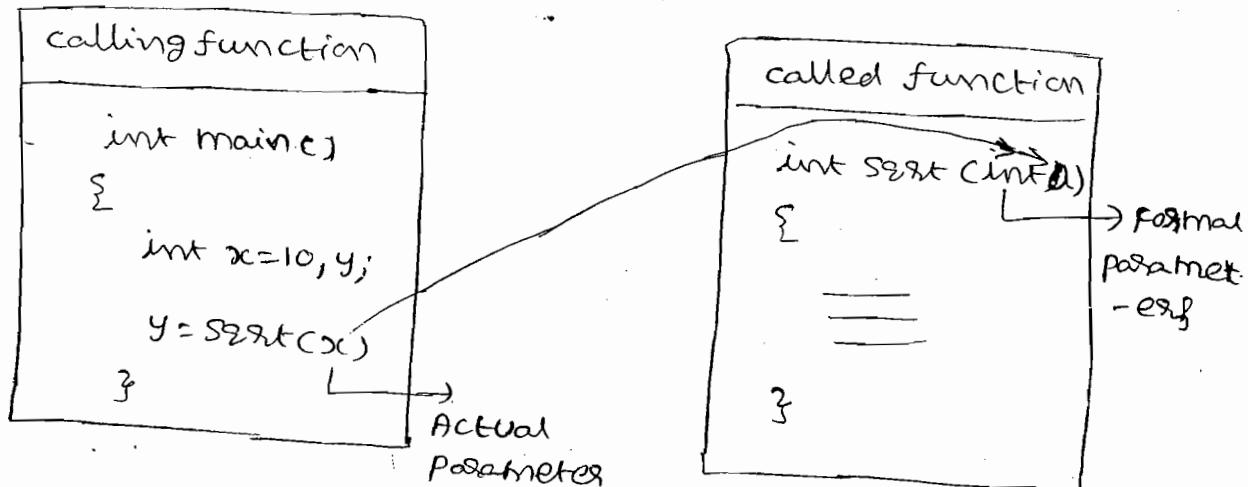
#### 1. Actual parameter :-

\* The data or values send from calling functions to called function are called Actual parameters.

#### 2. formal parameter :-

\* function parameters are called formal.

-parameters which receive values from calling function.



④ Function call :-

c++ supports three methods of calling function

1. call by value
2. call by address.
3. call by reference.

\* Function is divided into two parts

1. Function declaration/prototype.
2. Function definition.

Q) What is function prototype?

- A)
- \* Declaration of function is called of function prototype.
  - \* which define the properties of functions.
  - \* function prototype does not have body. called it header.

int max(int, int);

Q) What is function definition?

A) Function definition having function header + body (contents)

int max(int, int); → This is function declaration.

int max(int, int){  
} } → This is function definition.

- \* Function is declared in 4 types.
  - 1. Function with return type with parameters.
  - 2. Function with return type without parameters.
  - 3. Function without return type with parameters.
  - 4. Function without return type without parameters.

Program :

// function without return type with all parameters.

#include <iostream.h>

#include <conio.h>

Void draw-line();

int main()

{

clrscr();

draw-line();

cout << "In C++";

draw-line();

cout << "In object oriented";

draw-line();

return 0;

}

Void draw-line()

{

int i;

cout << "In";

for (i=1; i<=40; i++)

cout << "\*";

}

O/P

\*\*\*\*\* \* \*\*\*\*\*

C++

\*\*\*\*\* \* \*\*\*\*\*

object oriented

\*\*\*\*\* \* \*\*\*\*\*

- \* On invoking or a function execution control switched from calling function to called function after processing called function, the control return back to calling function.

~~Function~~ return :-

return is a keyword exits immediately from the currently executing function to the calling routine, optionally returning a value.

Syntax:-

return [expression];

Ex :-

@ Find output ?

Void print C)

{

cout << "Hello",

O/P

return;

Hello

cout << "Bye",

}

int main C)

{

print ( ),

return 0;

}

Program :-

// Function with parameters with return value

#include <iostream.h>

#include <conio.h>

int max (int x, int y)

{

if ( $x > y$ )

    return  $x$ ;

else

    return  $y$ ;

}

O/P

maximum is 40

int main ()

{

    class C;

    int big;

    big = max(20, 40);

    cout << "The maximum is " << big;

    return 0;

}

Q what is call by value?

A \* Calling or invoking function by sending constants  
or Variables

\* In call by value, Actual parameters are copied inside formal parameters.

\* If any change in formal parameters that does not reflect to actual parameters.

#include <iostream.h>

#include <conio.h>

Void Swap (int x, int y)

{

    int z;

    z = x;

    x = y;

    y = z;

}

```

int main()
{
    int a, b;
    class C;
    cout << "In input a,b values";
    cin >> a >> b;
    swap(a, b);
    cout << "In After swapping In";
    cout << "In a = " << a;
    cout << "In b = " << b;
    return 0;
}

```

O/P  
Input ~~a, b~~ a, b values  
10  
20  
After swapping  
a = 10  
b = 20

- \* Actual parameters can not changed inside called functions.

call by Address :-

- \* calling or invoking function by sending address of variable or address
- \* In call by Address Actual parameters can be modified in called functions
- \* call by Address allows to share data between functions.
- \* In call by Address function formal parameters are declared with pointer type

Program :

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

void swap (int *x, int *y)
{
    int temp,
        temp = *x,
        *x = *y,
        *y = temp;
}

int main ()
{
    int a, b;
    clrscr();
    cout << "Input a, b values",
    cin >> a >> b;
    swap (&a, &b);
    cout << "After swaping \n";
    cout << "a=" << a;
    cout << "b=" << b;
    return 0;
}
```

O/P  
Input a, b values

10

20

After swaping

a = 20

b = 10

Program :

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

void string-copy (char *dest, char *src)
{
    for (int i=0; src[i] != '\0'; i++)
        dest[i] = src[i];
```

dest[i] = 'o';

}

int main()

{

char str1[10], str2[10];

clrscr();

cout << "In input any string";

cin >> str1;

string - copy (str2, str1);

cout << "In string1: " << str1;

cout << "In string2: " << str2;

return 0;

}

O/P

input any string  
Bala

string1 = bala

string2 = bala

Program :

#include <iostream.h>

#include <conio.h>

Void print (char \*s)

{

cout << s;

}

int main()

{

clrscr();

print ("Hello");

return 0;

}

O/P

Hello

call by Reference :-

This is new feature is added in c++

- \* In call by reference function is called invoked by sending variables.
- \* In call by reference called function parameters are declared type of references.
- \* It allows to share data between functions.

Q @ Difference between Deep copy and Shallow copy?

A Deep copy involves using the contents of one object to create another instance of the same class.

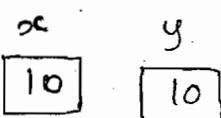
In Deep copy, the two objects (variables) may contain the same information but the target object will have its own buffers and resources. The destruction of either object will not affect the remaining object.

Shallow copy involves copying the contents of one object into another instance of the same class thus creating a mirror image. Owing to straight copying of references and pointers, the two objects will share the same externally contained contents of the other object.

Deep copy

int x = 10;

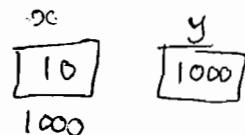
int y = x;



Shallow copy

int x = 10;

int \*p = &x;



### Program :-

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

Void Swap (int &x, int &y) /* function parameters
{
    x=x+y;
    y=x-y;
    x=x-y;
}

int main()
{
    int a, b;
    clrscr();
    cout<<"\n Input a, b values";
    cin>>a>>b;
    Swap (a, b);
    cout<<"\n a = " << a;
    cout<<"\n b = " << b;
    return 0;
}
```

### Inline function :-

- \* C++ is supported by inline function, but not supported in 'C'
- \* Every function is have own memory or address.

### Function :-

\* Every time a function is called, It takes a lot of extra time in executing a series of instructions like jumping to the function, pushing arguments to the stack and returning to

\* when a function is small a substantial percentage of execution time may be spent in overheads.

\* To Eliminate the cost of calls to small functions. The solution is inline function.

Inline function :-

\* An inline function is declared with the keyword `inline`.

\* An inline function is a function that is expanded inline when it is invoked.

\* The compiler replaces the function call with corresponding function code.

Syntax

Inline function - header

{

function body

}

Program8.cpp

```
inline void display()
{
    cout << "I'm Inside In the
          function";
}

int main()
{
    display();
    display();
    return 0;
}
```

com  
piler

program1.obj

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

```
prog1.exe
int main()
{
}
```

(Q) will the inline function be compiled as the inline functions always? Justify.

(A) An inline function is a request and not a command. Hence it won't be compiled as an inline function always.

Explanation :

Inline-expansion could fail if the inline function contains loops, the address of an inline function is used or an inline function is called in a complex expression. The rules for inlining are compiler dependent.

\* If it returns any value and it the inline function is recursive

Ex:- program:

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

inline void display()
{
    cout<<"In Hello";
}

int main()
{
    clrscr();
    display();
    display();
    return 0;
}
```

O/P  
Hello.  
Hello.

## Function with Default Arguments :-

- \* C++ allows to call a function without specifying all its arguments.
- \* The function assigns a default value to the parameter, which does not have a matching argument in the function call.
- \* Default values are specified when the function is declared.

## Prototype of the Function with Default values :-

Ex :-

float value (float, int, float = 0.12);

- \* Here the prototype declares a default value of 0.12 to the third parameter.
- \* Function call like amount = value (5000, 7);
- \* Passes the value 5000 to the first and 7 to the second parameter. The function uses the default of 0.12 to third parameter.

Note :-

Default arguments are specified from right to left.

Ex :- 1. void function1 (int x=10, int y) → Invalid  
  {

  }

2. void function1 (int x=10, int y=20) → Valid.  
  {

Ex :-

```
int main()
{
    int x=10, y=20;
    function (x,y);
}

void function() (int p, int q)
{
    cout << p; → 10
    cout << q; → 20
    return 0;
}
```

\* The arguments which are send from calling function to called function evaluated from right to left and sending is done from left to right. It uses stack.

Ex

```
int main()
{
    int x=5;
    function (x++, x++);
}

void function (int a, int b)
{
    cout << a; → 6
    cout << b; → 5
    return c;
}
```

O/p

6

5

\* Default arguments can not support with Java  
But support .Net.

Ex :-

```
int main ()  
{  
    int x;  
    x = function1(c);  
    cout << "x = " << x;  
}  
  
int function1 (c)  
{  
    int a=5;  
    return a++, a++, a++;  
}
```

O/P  
7

\* Evaluation of return is done from left to right  
and the value is return from right to left.

\* Default must be added from right to left.

\* Default value to a particular argument in the middle  
cannot be provided.

Ex: int sum (int =5, int), is illegal.

Program :

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

```
float simple - interest (float amt, int, float rate=1.5)  
{
```

```
    return (amt * t * rate / 100);
```

```
}
```

```
int main ()
```

```
{
```

```
    clrscr();
```

O/P  
[ simple Interest is 1080  
simple Interest is 3840 ]

```

float si1 = simple_interest(6000, 12);
cout << "In simple Interest is " << si1;
float si2 = simple_interest(8000, 24, 2.0);
cout << "In simple Interest is " << si2;
return 0;
}

```

- \* A function having default argument is having polymorphic behaviour.
- 

### Function overloading :-

- \* Defining more than one function with same name by changing 1. no. of parameters  
2. type of parameters.  
3. order of parameters.
- \* Function overloading allows:
  1. Reusability.
  2. Extensibility
  3. creating hierarchy of the function provide similar operations with same name.
- \* In C++ we can use the same function name to create functions that perform a variety of different tasks.
- \* we can design a family of functions with one function name but with different argument lists.
- \* The function would perform different operations depending on the argument list in the function call.

Q @ what is name mangling?

A Name mangling is the process through which your C++ compiler give each function in your program a unique name. In C++, all programs have at least a few functions with the same name. Name mangling is a concession to the fact that linker always insists on all function names being unique.

Ex

```
int max( int x, int y )  
{  
    if (x > y)  
        return x;  
    else  
        return y;  
}  
  
int max( int x, int y, int z )  
{  
    if (x > y && x > z)  
        return x;  
    else  
        if (y > z)  
            return y;  
        else  
            return z;  
}  
  
int main()  
{  
    clrscr();  
    cout << max(10,20);  
    cout << max(30,40,20);  
}
```

Program :

// Example for Function overloading

```
#include <iostream.h>
```

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

```
float find-area (float r)
```

```
{
```

```
    return 3.147 * r * r;
```

```
}
```

```
float find-area (float b, float h)
```

```
{
```

```
    return 0.5 * b * h;
```

```
}
```

```
int main ()
```

```
{
```

```
    clrscr ();
```

```
    float area1 = find-area (1.5);
```

```
    float area2 = find-area (2.0, 2.5);
```

```
    cout << "Area of circle" << area1;
```

```
    cout << "Area of triangle" << area2;
```

```
    return 0;
```

```
}
```

O/P

Area of circle 7.08075

Area of triangle 2.5

Q) what is binding ?

A) linking function call with function body is called binding

Q) what is early binding / static binding / compile time binding ?

A) linking function call with function based on function name, no. of parameters, types of parameters or order of parameters by

Program :

```
#include <iostream.h>
#include <conio.h>
int Sqr (int n)
{
    return n*n;
}
float Sqr (float n)
{
    return n*n;
}
int main ()
{
    clrscr ();
    int S1 = Sqr (3),
        float S2 = Sqr (5.5),
        cout << "In Sqr of integer" << S1,
        cout << "In Sqr of float" << S2,
        return 0;
}
```

The compilation of above program C++ compiler gives an error.

Ambiguity between 'Sqr (int)' and 'Sqr (float)'  
↓  
(Means duplication)

Because there is an implicit conversion between integer and float.

Ex:-    float x,  
          x=10;  
          int x,  
          x=15;

program :

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

Void swap (int &x, int &y) // voidswap(int x, int y)
{
    x = x+y;
    y = x-y;
    x = x-y;
}

Void Swap (int *x, int *y)
{
    *x = *x + *y;
    *y = *x - *y;
    *x = *x - *y;
}

int main ()
{
    clrscr();
    int a=10, b=20;
    swap(a, b);
    cout<<"\n After swaping \n";
    cout<<"\n a=" <<a;
    cout<<"\n b=" <<b;
    swap (&a, &b);
    cout <<"\n a=" <<a;
    cout <<"\n b=" <<b;
    return 0;
}
```

After swaping

a=20  
b=10  
a=10  
b=20

Q/P

\* From the above program void swap (int &x, int &y)

After including above function ~~compiles~~ <sup>TO RUN</sup> program

By compiling the program C++ compiler displays an error

Ambiguity between 'swap(int, int)' and 'swap(int &, int &)' because we call with the function using same type.

\* In call by value function is called by sending the variable. and the call by reference function is called by sending the variable. In both types the function is used called using same syntax because of that the compiler gives an error.

② Find output ?

void function (float x),

{

cout << "x=" << x;

}

int main(),

{

function(10);

function(1.5);

return 0;

}

O/P  
[  
x=10  
x=1.5  
]

## Structures in C++

Q what is difference between structure in C and structure in C++?

A Structure in C

- \* collection of variables or data

- \* The contents of structure is global/public

- \* In both the language the structure is called user defined data type.

Structure in C++

- \* collection of variables and functions

- \* contents of structure is private, protected or public.

Q what is difference between structure in C++ and class in C++

Structure in C++

- \* collection of variables and functions

- \* contents of structures is private, protected and public

- \* Default content of structure is public

- \* not inherited

Class in C++

- \* collection of variables and functions

- \* contents of structure is private, protected and public

- \* Default content of class is private

- \* Inherited

\* In C++ The structures are used to build simple data types. The complex data types are used to develop class.

Struct :-

\* Struct is a keyword.

\* Groups variables into a single record.

Syntax

Struct structure-type-name

{

type variable-name;

type variable-name;

type variable-name;

:

};

Struct structure-type-name

{

type variable-name;

type variable-name;

type variable-name;

return-type function-name ([parameters])

{

};

Ex:- struct data

{

int dd, mm, yy;

};

### Syntax

" Accepting contents of structure

Structure-variable-name.variable-name

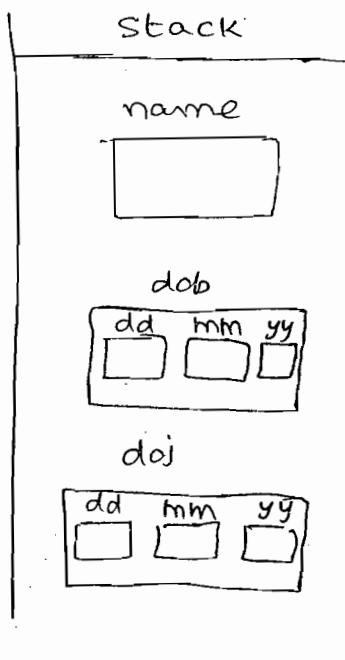
Here .(dot) is a member access operator or field access operator.

Program:

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

struct date
{
    int dd, mm, yy;
};

int main()
{
    char name[20];
    date dob, doj;
    clrscr();
    cout << "Input name";
    cin >> name;
    cout << " Input date of birth";
    cin >> dob.dd >> dob.mm >> dob.yy;
    cout << " Input date of Joining";
    cin >> doj.dd >> doj.mm >> doj.yy;
    cout << " Name " << name;
    cout << " Date of Birth " << dob.dd << "/"
        << dob.mm << "/" << dob.yy;
    cout << " Date of Joining " << doj.dd << "/"
        << doj.mm << "/" << doj.yy;
    return 0;
```



classes and objects :-

- \* The most important feature of C++ is the 'class'.
- \* It is a new way of creating and implementing a user-defined data type.
- \* A class is an extension of structure used in C.
- \* A class is a way to bind data and its associated functions together.
- \* It allows the data to be hidden.
- \* A class is a collection of members,
  1. data members,
  2. member functions.
- \* class is an implementation of Encapsulation.
- \* A class is a container which contains data and related functions.
- \* In C++ program is a collection of classes.
- \* In C program is a collection of functions.

The general form of class declaration is:

```
class class name  
{  
    private:  
        Variable declaration;  
        function declaration;  
    public:  
        Variable declaration;  
        function declaration;  
};
```

The function and variables are collectively called as members.

- \* The members declared as private can be accessed only from within the class.
- \* Public members can be accessed from outside the class.
- \* The key feature of data hiding of object oriented programming is achieved by private declaration.
- \* The use of keyword private is optional.
- \* By default all the members are private.
- \* The variables declared inside the class are known as data members.
- \* The functions declared inside the class are known as member functions.
- \* The binding of data and functions together into a single class type is known as Encapsulation.

```
class class  
{  
    int num1, num2;  
};
```

```
int main ()  
{  
    calc calc1;  
    ↓      ↓  
    class   object  
    calc1.num1 = 10; }      This is Error  
    calc1.num2 = 20; }
```

```
calc calc2;  
calc2.num1 = 10; }      This is Error  
calc2.num2 = 20; }
```

#### \* Creating objects :-

\* Once a class has been declared the variables of that type can be created

Ex: item x;

\* The above statement creates a variable x of type item

\* In C++ class variables are known as objects

\* One creation of object memory reserved ~~or allocated~~ for data members of a class.

\* In C++ private data is operated by member functions, friend functions and pointers

#### Accessing class members :-

\* The private data of a class is accessed only through member functions of that class

\* The format for calling a member function is  
object name.function-name (arguments),

- \* The member function is invoked only by using the object of the same class.

Ex :-

```

class calc
{
private:
    int num1, num2; } Data members
public:
    void add ()
    {
        int s = num1 + num2;
    }
    void sub ()
    {
        int s = num1 - num2;
    }
}

int main ()
{
    calc calc1;
    calc1.add ();
    calc1.sub ();
}
  
```

} member functions

$\downarrow$        $\downarrow$   
class      object / variable

- \* member functions operates on one or more than one object.
- \* The memory for member function are not allocated when the object is created.
- \* member functions are called on objects to operate data.

## Defining member functions

member functions can be defined in two places.

→ outside the class.

→ Inside the class.

Irrespective of the place of definition the function performs the same task.

### Inside the class

① Inline function is expanded during compile time.

② decrease Efficiency of program

### outside the class

① does not become inline.

② does not decrease Efficiency of program.

### Program:

// write a program to find area of triangle

```
#include <iostream.h>
```

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

```
class triangle
```

```
{
```

```
    float base, height;
```

```
public:
```

```
    void read();
```

```
{
```

```
    cout << "Input base, height";
```

```
    cin >> base >> height;
```

```
}
```

```
    float find_area();
```

```
{
```

```
    return 0.5 * base * height;
```

```
}
```

```
};
```

### Output

Input base, height

```

int main()
{
    triangle t1;
    clrscr();
    t1.read();
    float area1 = t1.find-area();
    cout << "In. Area of triangle is " << area1;
    return 0;
}

```

Note:-

A class does not have any data members occupying one byte space.

Defining the member function outside the class:

The general form of member function definition,

```

Returntype classname::functionname(arguments)
{
    Statement;
}

```

By defining the definition outside the class inline expansion is avoided.

Program:

// write a program to find area of circle

```

#include <iostream.h>
#include <conio.h>
class circle
{
    float r;
public:

```

```

Void read-r ();
float find-area ();
};

Void circle:: read-r()
{
    cout << "In input r";
    cin >> r;
}

float circle:: find-area()
{
    return 3.14159 * r * r;
}

int main()
{
    circle circle1;
    circle1.read-r();
    float area = circle1.find-area();
    cout << "Area of circle is " << area;
    return 0;
}

```

Output

Input r 3.0

Area of circle is

member function with parameters :-

- ② Define a way other than using the keyword `inline` to make a function `inline`?
- Ⓐ The function must be defined inside the class.
- \* a member function having parameters receive values from another function (calling function)

\* An object communicate with another object by sending values, which called message passing.

1. message sender. (This is object)
2. message receiver. (This is object)
3. message (Function)

Q Difference between the message and method ?

A

message

- \* objects communicate by sending messages to each other.
- \* A message is sent to invoke a method.

method

- \* provides response to a message
- \* It is an implementation of an operation.

program :

// member function having parameters.

```
#include <iostream.h>
```

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

```
class date
```

```
{
```

```
    int dd, mm, yy;
```

```
public:
```

```
Void set-date(int d, int m, int y)
```

```
{
```

```
    dd = d;
```

```
    mm = m;
```

```
    yy = y;
```

```
}
```

```
Void print-date()
```

```
{
```

```
cout << "In" << dd << "It" << mm << "It" << yy;
```

```
{
```

```
.
```

```

int main()
{
    date date1;
    class();
    date1.set_date(10,10,2009);
    date1.print_date();
    return 0;
}

```

DATE  
 O/P  
 10 10 2009

Q) what is modifier?

A) A modifier, also called a modifying function is a member function that changes the value of at least one data member. In other words, an operation that modifies the state of an object. Modifiers are also known as 'mutators'.

Q) what is an Accessor?

A) An accessor is a class operation that does not modify the state of an object. The accessor functions need to be declared as constant operations.

\* An object whose state can be changed <sup>is called</sup> ~~is mutable~~ object.

\* An object whose state cannot be changed is called immutable object.

Program :

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

class Student
{
    char name[20];
    char course[20];
public:
    void set_data(char n[], char c[]);
    void print_data();
};

void Student::set_data(char n[], char c[])
{
    strcpy(name, n);
    strcpy(course, c);
}

void Student::print_data()
{
    cout << "In Name " << name;
    cout << " In Course " << course;
}

int main()
{
    Student stud1;
    stud1.set_data("Rama", "C++");
    stud1.print_data();
    stud1.set_data("Bala", "Oracle");
    stud1.print_data();
    return 0;
}
```

class A

{

```
int x=10; //invalid  
int y=20; //invalid  
}
```

\* data members of a class can not be initialized with in the class directly. By assigning values.

Program :

// member function with parameters and return type

```
#include <iostream.h>
```

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

class rectangle

{

```
float l, b;
```

public:

```
Void set-l (float);
```

```
Void set-b (float);
```

```
float find-area();
```

}

```
Void rectangle::set-l (float p)
```

{

```
l=p;
```

}

```
Void rectangle::set-b (float q)
```

{

```
b=q;
```

}

```
float rectangle::find-area()
```

{

```
return l*b;
```

}

```

int main()
{
    rectangle rect1;
    classA();
    rect1.set_l(2.5);
    rect1.set_b(2.0);
    float area = rect1.find_area();
    cout << "In Area is " << area;
    return 0;
}

```

O/P

Area is 5

~~QUESTION~~  
// members functions with class type parameters :-

- \* A member function class type parameters receive objects.
- \* one object can be send to another object using member function with class type parameters.
- \* It allows to operate more than one object.

Program:

```

#include <iostream.h>
#include <conio.h>
class A
{
    int x,y;
public:
    void read_A()
    {
        cout << "In Input x,y Values";
        cin >> x >> y;
    }
}

```

```

Void compare (A obj)
{
    if (x == obj.x && y == obj.y)
        cout << "The objects are equal";
    else
        cout << "The objects are not equal";
}
;

int main()
{
    A obj1, obj2;
    cin >> obj1;
    obj1.read_A();
    obj2.read_A();

    implicit ← obj1.compare (obj2);
    return 0;           ↘ Explicit
}

```

---

\* member function having return type of class, it return object

Program :

// write a program to add two complex numbers

```

#include <iostream.h>
#include <conio.h>
class complex
{
    float real, img;
public:
    Void read_complex ();
    Void print_complex ();
    complex add_complex (complex);

```

```

Void complex :: read-complex ( )
{
    cout << "In Input complex numbers ";
    cin >> real >> img;
}

Void complex :: print-complex ( )
{
    cout << "In " << real << "i" << img;
}

complex complex :: add-complex (complex c2)
{
    complex c3;
    c3.real = real + c2.real;
    c3.img = img + c2.img;
    return c3;
}

int main ( )
{
    complex comp1, comp2, comp3;
    clrscr();
    comp1.read-complex ();
    comp2.read-complex ();
    comp3 = comp1.add-complex (comp2);
    comp3.print-complex ();
    return 0;
}

```

output:

Input complex numbers,

1.2  
1.5

Input complex numbers,

1.5  
1.8

program:

write a program to add two matrices

```
#include <iostream.h>
#include <conio.h>
class matrix
{
    int a[3][3];
public:
```

```
    void read_matrix();
```

```
    void print_matrix();
```

```
    matrix add_matrix(matrix);
```

```
};
```

```
void matrix::read_matrix()
```

```
{
```

```
    cout << "Input elements of matrix";
```

```
    for (int i=0; i<3; i++)
```

```
        for (int j=0; j<3; j++)
```

```
            cin >> a[i][j];
```

```
}
```

```
void matrix::print_matrix()
```

```
{
```

```
    for (int i=0; i<3; i++)
```

```
{
```

```
        for (int j=0; j<3; j++)
```

```
            cout << " " << a[i][j],
```

```
            cout << "\n";
```

```
}
```

```
}
```

```

matrix matrix:: add_matrix(matrix m2)
{
    matrix m3;
    for (int i=0; i<3; i++)
        for (int j=0; j<3; j++)
            m3.a[i][j] = a[i][j] + m2.a[i][j];
    return m3;
}

int main()
{
    matrix matrix1, matrix2, matrix3;
    clrscr();
    matrix1.read_matrix();
    matrix2.read_matrix();
    matrix3 = matrix1.add_matrix(matrix2);
    matrix3.print_matrix();
    return 0;
}

```

Output :

Input elements of matrix

1 2 3

4 5 6

7 8 9

Input elements of matrix

4 5 6

7 8 9

1 2 3

5 7 9

11 13 15

8 10 12

## Friend Functions :-

A non-member function can not have access to the private data of a class.

There could be a situation where two classes have to share a particular function. In such situations C++ allows a common function made friendly with both the classes.

Ex : class test

```
{  
public : friend void tax();  
};
```

- \* A friend function declaration must be preceded by the keyword friend.
- \* It is not in the scope of the class to which it has been declared as friend.
- \* It is not called using the object of that class.
- \* It can be invoked like a normal function without any object.
- \* It can not access the member names directly.
- \* It has to use an object name dot membership operator and member name.
- \* It can be declared either in the public or private part of the class without affecting its meaning.
- \* Usually it has the objects as arguments.
- \* A friend function can be called by reference. It should be used only when absolutely necessary, because such function alters the values of the private members which is against data hiding.

\* Friend functions are often used in operator overloading.

Class A

{

int x,y;

public:

friend void print(A obj),

}

Class B

{

int p,q;

public:

friend void print(A obj),

}

- void print(A obj1, B obj2)

{

body of friend function

}

Program :

#include <iostream.h>

#include <conio.h>

Class A

{

int x,y;

██████████

friend void print(A obj);

public:

void print(A obj)

{

cout << obj.x << " " << obj.y;

}

int main()

{

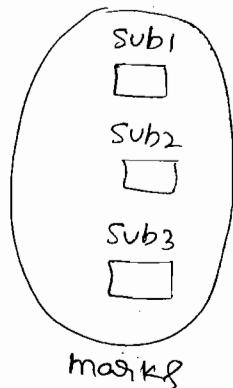
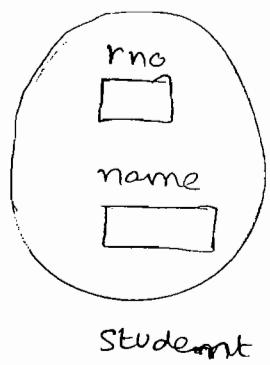
A obj1;

clrscr();

print(obj1);

return 0;

}



find result (student, marks);

}

~~~~~  
program :

```

#include <iostream.h>
#include <conio.h>

class marks;
class student
{
    int rno;
    char name[20];
public:
    void read_student();
    friend void
    find_result(student, marks);
};

class marks
{
    int sub1, sub2, sub3;
public:
    void read_marks();
    friend void
    find_result(student, marks);
};
  
```

```
void Student::read_student()
{
    cout << "Input rollno";
    cin >> rno;
    cout << "Input name";
    cin >> name;
}

void Marks::read_marks()
{
    cout << "Input 3 subject marks";
    cin >> sub1 >> sub2 >> sub3;
}

void find_result(Student s, Marks m)
{
    cout << "Rollno" << s.rno;
    cout << "Name" << s.name;
    if(m.sub1 <= 40 || m.sub2 <= 40 || m.sub3 <= 40)
        cout << "Result is fail";
    else
        cout << "Result is pass";
}

int main()
{
    Student stud1;
    Marks final;
    Class cl;
    stud1.read_student();
    final.read_marks();
    find_result(stud1, final);
    return 0;
}
```

Program :

// comparing two different types of objects.

#include <iostream.h>

#include <conio.h>

class B;

class A

{

int x, y;

public:

Void read -AC,

{

cout<<"In Input x,y values",

cin>>x>>y;

}

friend Void compare (A o1, B o2),

,

class B

{

int p, q;

public:

Void read -BC,

{

cout<<"In Input p,q values",

cin>>p>>q;

}

friend Void compare (A o1, B o2),

,

Void compare (A o1, B o2)

{

if (o1.x == o2.p && o1.y == o2.q)

```

cout << "In objects are equal";
else
    cout << "In objects are not equal";
}

int main()
{
    A obj1;
    B obj2;
    class();
    obj1.read(A());
    obj2.read(B());
    compare(obj1, obj2);
    return 0;
}

```

Q what is difference between member function and friend function ?

A

member function

- \* belongs to one class/  
Same class
- \* operates on similar type  
of objects
- \* It is bind with object
- \* It is called using object  
name

friend function

- \* not a member of any  
class.
- \* operates on different type  
of objects
- \* Not bind with any object
- \* It is called like normal  
functions

## friend class :-

A friend of a class X is a function or class that, although not a member of that class, has full access rights to the private and protected members of class X.

### Syntax:

friend < identifier >;

- \* identifier is nothing but class name

## contained class :-

An existing class is called contained class.

### Contained class :

A class which hold object of contained class is called contained class.

- \* creating object of one class inside another class is called composition.
- \* composition allows class reusability.
- \* when an object of one class is created in another class the other class is able to access the public members of contained class.
- \* In order to access private members of contained class contained class is declared as a friend.
- \* composition does not allow to extend the functionality of contained class. It allows only reusability.

program :

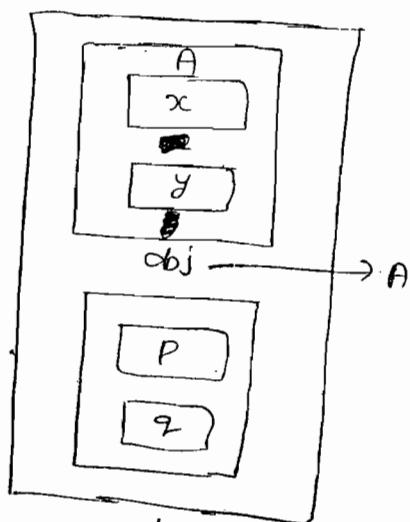
```
#include <iostream.h>
#include <conio.h>
class A
{
    int x,y;
friend class B;
};

class B
{
    A obj,
    int p,q;
public:
    void print()
    {
        cout<<"\n" <<obj.x << "\t" << obj.y,
        cout<<"\n" << p << "\t" << q;
    }
};

int main()
{
    B obj1;
    class C;
    obj1.print();
    return 0;
}
```

output:

garbage values are  
printed because no assign to the  
values.



program:

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

class author
{
    char a_name[20];
    friend class book;
};

class book
{
    author author_details;
    char b_name[20];
    float price;
public:
    void read_book();
    void print_book();
};

void book::read_book()
{
    cout << "In Input book name";
    cin >> b_name;
    cout << "In Input author name";
    cin >> author_details.a_name;
    cout << "In Input price";
    cin >> price;
}

void book::print_book()
{
    cout << "In Book Name" << b_name;
    cout << "In Author Name" << author_details.a_name;
    cout << "In price of book" << price;
}
```

```

int main ()
{
    book book1;
    class1();
    book1.read_book();
    book1.print_book();
    return 0;
}

```

\*this pointer :-

- \*this holds an address of current object.
- Every non static member function of a class having local variable \*this.
- If local variable and data member with same name, data member is referred by using \*this.

Program :

```

#include <iostream.h>
#include <conio.h>

class A
{
    int x, y;
public:
    void set_data(int x, int y)
    {
        (*this).x = x,
        (*this).y = y;
    }
    void print_data(),
}

```

```

cout << x << y;
}

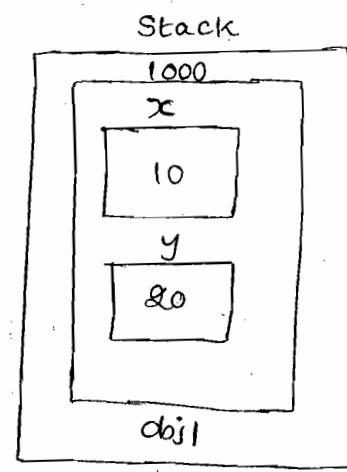
};

int main()
{
    A obj1;
    obj1.set_data(10,20);
    obj1.print_data();
    return 0;
}

```

---

program :



```

#include <iostream.h>
#include <conio.h>

class A
{
public:
    void print()
    {
        cout << "In" << this;
    }
};

int main()
{
    A obj1;
    cout << &obj1 << endl;
    obj1.print();
    return 0;
}

```

---

Program :

```
#include <iostream.h>
#include <conio.h>
class time
{
    int hh, mn, ss;
public:
    void set_time(int hh, int mn, int ss);
    void print_time();
};

void time::set_time(int hh, int mn, int ss)
{
    (*this).hh = hh;
    (*this).mn = mn;
    (*this).ss = ss;
}

void time :: print_time()
{
    cout << "In" << hh << ":" << mn << ":" << ss;
}

int main()
{
    time time_in;
    clrscr();
    time_in
    time_in.set_time(10, 40, 50);
    time_in.print_time();
    return 0;
}
```

Output  
10:40:50

## Constructors And Destructors

Constructors :-

Q) why constructors?

(i) class A

{

```
int x=10; }  
int y=20; }  
};
```

Invalid because of  
Data members can not  
initialize with in class

(ii) class A

{

```
int x;  
int y; }  
};
```

garbage values.

```
int main()
```

{

A obj1;

}

(iv) class A

{

```
int x,y;
```

};

```
int main()
```

{

A obj1 = {10, 20}; → Error

}

A obj1 = {10, 20};

A obj2 = {30, 40};

};

Data members are public while creating values  
defining.

## Constructors :-

- \* C++ provides a special member function, called constructor, which enables an object to initialize itself.
- \* This is known as automatic initialization of objects.
- \* C++ also provides another member function called destructor that destroys the objects.
- \* Constructors are used in two ways,
  1. Defining Initial State of object
  2. Allocating Resources to object.
- \* The constructor is invoked whenever an object of its associated class is created.
- \* It constructs the values of the data members of the class.

## Characteristics of Constructors :-

- \* They should be declared in the public section.
- \* They are invoked automatically when the objects are created.
- \* They do not have return type, not even void and they can not return values.
- \* They can not be inherited.
- \* Constructors can be overloaded.
- \* They can have default values for arguments.
- \* We can not refer to their addresses.
- \* They can make implicit calls to new and delete.

\* The constructor name is same as the class name.

Different types of constructors :-

These are three types.

- (i) Default constructor.
- (ii) Parameterized constructor.
- (iii) Copy constructor.

Default constructor :-

\* Non parameterized constructor is called default constructor.

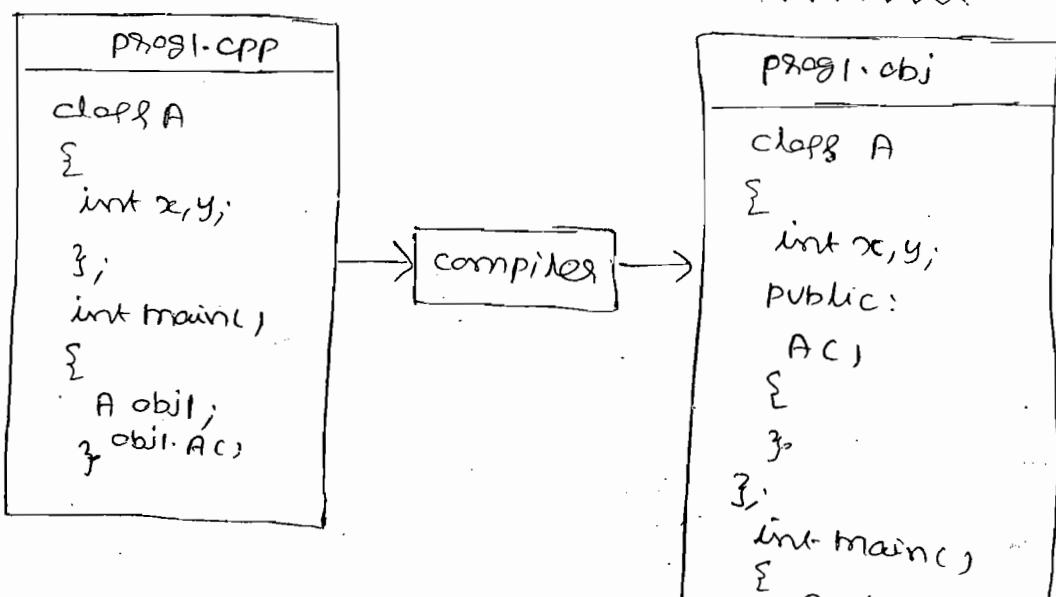
(OR)

\* A constructor with zero parameters is called default constructor.

\* There are two types

- ① compiler written (compiler defined)
- ② user defined

→ compiler defined default constructor



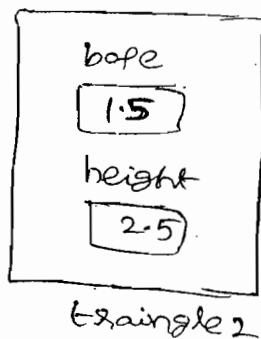
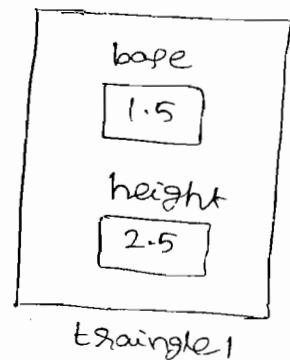
compiler defined default constructor.

- \* If there is no constructor with in class compiler writes a constructor with out any parameters and contents.

User defined default constructor :-

Program :-

```
#include <iostream.h>
#include <conio.h>
class triangle
{
    float base, height;
public:
    triangle()
    {
        base = 1.5;
        height = 2.5;
    }
    int main()
    {
        triangle triangle1;
        triangle triangle2;
        return 0;
    }
}
```



note :

- \* constructor is invoked only once on object.
- \* constructor is not used calling dot operator (.)

parameterized constructor :-

- \* A constructor which receive values from outside.
- \* This constructor initialize object with different states.

program :

```
#include <iostream.h>
#include <conio.h>
class triangle
{
    float base, height;
public:
    triangle (float b, float h);
    float find-area();
};

triangle :: triangle (float b, float h)
{
    base = b;
    height = h;
}

float triangle :: find-area()
{
    return 0.5 * base * height;
}

int main()
{}
```

```
triangle triangle1;
```

```
triangle1 *triangle2; // → Error.
```

\* Because there is no default constructor with  
in the class.

```
triangle triangle1(1.5, 2.0);
```

```
triangle triangle2(2.5, 3.0);
```

```
class C;
```

```
float area1 = triangle1.find_area();
```

```
float area2 = triangle2.find_area();
```

```
cout << "In Area of triangle 1:" << area1;
```

```
cout << "In Area of triangle 2:" << area2;
```

```
return 0;
```

```
}
```

Output

```
Area of triangle1: 1.5
```

```
Area of triangle2: 3.75
```

Example:

```
class A
```

```
{
```

```
int x, y;
```

```
public:
```

```
A(int x, int y)
```

```
{
```

```
(*this).x = x,
```

```
(*this).y = y,
```

} Initialization

```
}
```

```

void set( int x, int y )
{
    (*this).x = x;
    (*this).y = y;
}

void print()
{
    cout << x << y;
}

int main()
{
    A obj1(10, 20);
    obj1.print();
    obj1.set(30, 40);
    obj1.print();
}

```

Constructor overloading :-

- \* Defining more than one constructor with in class is called constructor overloading.
- \* Constructor is overloaded by changing no. of parameters, types of parameters, or order of parameters.

② Differences between member function and constructor?

member function

- \* Name can be anything
- \* It is called any no. of times of object
- \* Return type

constructor

- \* Name should be same as class.
- \* only once
- \* no return type

- \* overloaded constructor is having polymorphic behaviour.
- \* constructor is overloaded to extend functionality to the existing constructor.

Program: Example for constructor overloading

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

class circle
{
    float r;
public:
    circle();
    circle(float);
    float find-area();
};

circle::circle()
{
    r=2.0;
}

circle::circle(float r)
{
    (*this).r=r;
}

float circle::find-area()
{
    return 3.147 * r * r;
}

int main()
{
    circle circle1;
    circle circle2(3.0);
    class c;
    float area=circle1;
}
```

```
float area2 = circle2.find_area();
cout << "In Area of circle1:" << area1;
cout << "In Area of circle2:" << area2;
return 0;
```

{

Output:-

Area of circle1:

Area of circle2:

Program:-

```
#include <iostream.h>
```

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

```
#include <string.h>
```

```
class Student
```

{

```
char name[10];
```

```
char course[30];
```

```
float fees;
```

```
public:
```

```
Student (char n[], char c[])
```

{

```
strcpy(name, n);
```

```
strcpy(course, c);
```

```
fees = 0;
```

}

```
Student (char n[], char c[], float f)
```

{

```
Student::student(n, c);
```

```
fees = f;
```

}

```
void printStudent() {
    cout << "In Name:" << name;
    cout << "In course:" << course;
    cout << "In Fees:" << fees;
}
```

```
};
```

```
int main() {
```

```
{
```

```
    Student stud1("Bala", "C");
```

```
    Student stud2("Sita", "C++", 1000);
```

```
    class C;
```

```
    stud1.printStudent();
```

```
    stud2.printStudent();
```

```
    return 0;
}
```

Output:  
~~~~~:

Name: Bala

Course: C

Fees: 0

Name: Sita

Course: C++

Fees: 1000

\* constructor of one class is called within another  
constructor of same class using the following syntax

Syntax: class name :: constructor name

Program :-

Allocating memory for object.

```
#include <iostream.h>
#include <conio.h>
class vector
{
    int *a;
    int size;
public:
    Vector (int s)
    {
        size = s;
        a = new int [size];
    }
}
```

```
void read_elements ( )
{
    cout << "In Input elements ";
    for (int i=0; i<size; i++)
        cin >> a[i];
}

void print_elements ( )
{
    cout << "In Elements are: \n";
    for (int i=0; i<size; i++)
        cout << a[i];
}
```

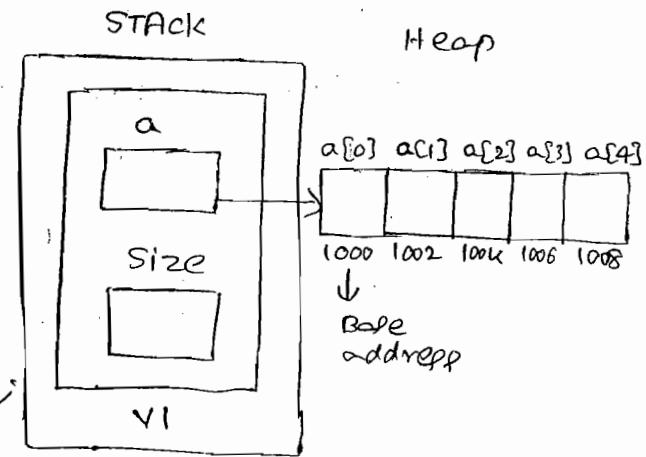
```
int main ( )
{
```

```
    class vector;
```

```
    Vector v1(5);
```

```
    v1.read_elements();
```

```
    v1.print_elements();
```



OUTPUT:

Input elements

10  
20  
30  
40  
50

Elements are

10 20 30 40 50

~~QUESTION~~

Copy constructor :-

- \* A copy constructor is used to initialize an object from another object.
- \* copy constructor is parameterized constructor having reference type parameters.
- \* This constructor receive another object in order to init current object.
- \* It is having parameters of type class.

The statement,

code obj2(obj1);

Defines and initializes the obj2 with obj1 values.

Ex :

A obj1(10,20); → obj1.A(10,20)

A obj2 (obj1); → obj2.A(obj1)

Program:

```
#include <iostream.h>
```

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

```
class A
```

```
{
```

```
int x,y;
```

```
public:
```

```
A (int x,int y)
```

```
{
```

```
(*this).x=x;
```

```
(*this).y=y;
```

```
}
```

```
A (A &obj)
```

```
{}
```

```

x = obj.x;
y = obj.y;
}

Void Print()
{
cout << " \n x = " << x;
cout << " \n y = " << y;
}

int main()
{
A obj1(10, 20);
A obj2(obj1);
class obj c;
obj1.print();
obj2.print();
return 0;
}

```

Output:

x=10  
y=20  
x=10  
y=20

Destructor :-

- \* A destructor destroys the objects that have been created by a constructor.
- \* A destructor is a member function whose name is same as the class name.
- \* A destructor name is preceded by tilde (~).
- \* A destructor never takes any arguments nor it returns any value.
- \* Destructor is not overloaded.
- \* The compiler invokes it implicitly when the program is terminated to clean up storage space.
- \* It is good practice to declare destructors since it provides consistency.

Syntax:

```
~destructor-name( )  
{  
    Statement;  
}
```

Program:

```
#include <iostream.h>  
#include <conio.h>  
class A  
{  
public:  
    A()  
    {  
        cout<<"In Inside constructor";  
    }  
    ~A()  
    {  
        cout<<"In Inside destructor";  
    }  
};  
int main()  
{  
    A obj;    clrscr();  
    return 0;  
}
```

Program :

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

class matrix
{
    int **a;
    int rsize, csize;
public:
    matrix(int rsize, int csize)
    {
        (*this).rsize = rsize;
        (*this).csize = csize;
        a = new int*[rsize];
        for (int i=0; i<rsize; i++)
            *(*(a+i)) = new int[csize];
    }

    void read_elements()
    {
        cout << "Input elements: ";
        for (int i=0; i<rsize; i++)
            for (int j=0; j<csize; j++)
                cin >> *(*((a+i)+j));
    }

    void print_elements()
    {
        cout << "Elements are: \n";
        for (int i=0; i<rsize; i++)
            for (int j=0; j<csize; j++)
                cout << *(*((a+i)+j)) << " ";
    }
}
```

```

cout << "m";
}

~matrix()
{
    for (int i=0; i<rsize; i++)
        delete *(cati);
    delete a;
}

int main()
{
    matrix matrix1(2, 2);
    clrscr();
    matrix1.read_elements();
    matrix1.print_elements();
    return 0;
}

```

Output :

Input elements :

2 3  
4 5

Elements are :

2 3  
4 5

---

Static members :-  
~~~~~

Static Data member:  
~~~~~

A data member of a class can be static. It is initialized to zero when the first object of the class is created. No other initialization is permitted.

only one copy of that member is created for the entire class and is shared by all objects of the class.

Syntax:  
~~~~~

```
class A  
{  
    int x, y;  
    static int z;  
};
```

\* Static data member is having  
(i) declared  
(ii) defined

\* Static data member is declared with in class and definition is given outside the class.

Syntax:  
~~~~~

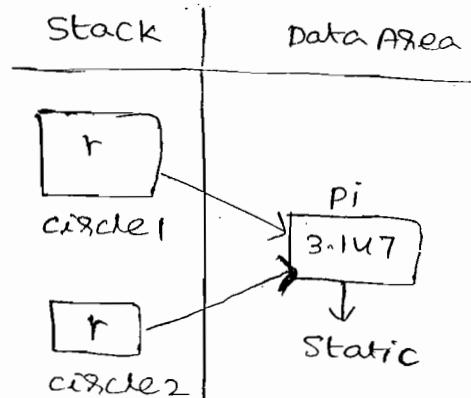
static datatype variable-name;

datatype class-name:: variable-name = [value];

\* Static data member's memory is allocated with in data area which is a global area shared by no. of objects.

program:

```
#include <iostream.h>
#include <conio.h>
class circle
{
    float r;
    static float pi;
public:
    void set_r (float r)
    {
        (*this).r = r;
    }
    float find_area ()
    {
        return pi * r * r;
    }
};
float circle::pi = 3.147;
int main ()
{
    circle circle1;
    circle circle2;
    clrscr ();
    circle1.set_r (2.0);
    circle2.set_r (3.5);
    float area1 = circle1.find_area ();
    float area2 = circle2.find_area ();
    cout << "In Area of circle1: " << area1;
    cout << "In Area of circle2: " << area2;
    return 0;
}
```



- \* static data member is accessed using class name

Syntax:

class-name :: member-name

Ex:-

class A

{

public:

int x;

static int y;

}

int A::y;

int main()

{

cout << A::y;

cout << A::x; → Error

A obj;

cout << obj.x;

cout << obj.y;

}

- Q what is difference between static data members and non static data members?

A

Static data member

\* It bind with class name and object name.

\* memory is allocated on creation of object or when accessed first time

\* more than one object having same static data members.

Non static data member

\* It bind with object name

\* memory is allocated on creation of object.

\* Each object having its own Data members

Static member function :-

- \* Static member function operates on static data but can not operate on non-static data members.
- \* Static member function is called/bind with class name.

Ques @

Describe the main characteristics of static functions?

(A)

The main characteristics of static functions include.

- \* It is without the a this pointer.
- \* It can't directly access the non static members of its class.
- \* It can't be declared constant, volatile or virtual.
- \* It doesn't need to be invoked through an object of its class, although for convenience, it may.

Ex :-

```
class A
{
    int x;
public:
    void set (int x)
    {
        (*this).x = x;
    }
    static void set (int x)
    {
        (*this).x = x; → Error
    }
}
```

Program :

```
#include <iostream.h>
#include <conio.h>
class math
{
public:
    static int max(int x, int y)
    {
        if(x > y)
            return x;
        else
            return y;
    }
};

int main()
{
    int m;
    m = math::max(30, 40);
    cout << "Max is " << m;
    return 0;
}
```

Constant Members :- These are two types.

1. Constant data members.
2. Constant member functions.

1. Constant data members :-

\* Constant data members values can not be changed.

\* It is used to define constant state of

Syntax :

~~~~~

const datatype Variable-name;

\* constant data members are initialize using const-  
-uctor list/ constructors.

Program :

```
#include <iostream.h>
#include <conio.h>
class Simple-interest
{
    float amt;
    int time;
    const float rate;
public:
    Simple-interest(): rate(2.0)
    {
    }
    void set(float amt, int time)
    {
        (*this).amt = amt;
        (*this).time = time;
    }
    float find-interest()
    {
        return (amt * time * rate) / 100;
    }
};
```

Output :

```

int main()
{
    Simple-interest si;
    class();
    si.set(8000, 12);
    float si = si.find_interest();
    cout << "In Simple Interest is: " << si;
    return 0;
}

```

- \* The value of constant data member cannot be changed by member functions.

Constant member functions :-

- \* Constant member function provides read only operations.
- \* This function can not change state of object.
- \* This function can not modify the values of data members.
- \* Constant member function is called as An Accessor member function.

Syntax :-

```

return-type function-name (parameters-list)
    const
{
    statements;
}

```

Program :

```
#include <iostream.h>
#include <conio.h>
class Student
{
    int rno,
        sub1, sub2, sub3;
public:
    void read();
    {
        cout << "In Input roll no",
        cin >> rno;
        cout << "In Input 3 subjects",
        cin >> sub1 >> sub2 >> sub3;
    }
    void display() const
    {
        cout << "In Rollno" << rno;
        cout << "In Total " << sub1 + sub2 + sub3;
        cout << "In Avg " << (sub1 + sub2 + sub3) / 3;
    }
};

int main()
{
    Student stud1;
    clrscr();
    stud1.read();
    stud1.display();
    return 0;
}
```

## Operator overloading :

- \* The mechanism of giving special meanings to an operator is known as operator overloading.
- \* This is one of the existing features of C++.
- \* This technique has enhanced the power of extensibility of C++.
- \* This technique permits to add two user-defined variables with the same syntax that is applied to basic types.
- \* Provides the new definitions for most of the C++ operators.
- \* Operator is overloaded in order to operate user-defined datatype.
- \* Existing operators operates on predefined datatypes.
- \* Overloaded operator is a special function.
- \* We can overload all the C++ operators except class member access operator (., .\*)  
scope resolution operator (::)  
size of operator (sizeof)  
conditional operator (? :)

## Defining operator overloading

```
return type class name :: operator op(arg-list)
{
    function body
}
```

another syntax :-

```
operator <operator symbol>(<parameters>)
{
    <Statement(s)>;
}
```

The keyword "operator", followed by an operator symbol, defines a new (over loaded) action of the given operator.

operator is defined a new action. And operator is a keyword.

Rules for overloading operators :-

only existing operators can be overloaded

The operator must have at least one operand of user-defined type.

New operators can not be created.

We can not change the basic meaning of an operator.

operators can not be overridden.

When binary operators overloaded through member function, the left hand operand must be an object of the relevant class.

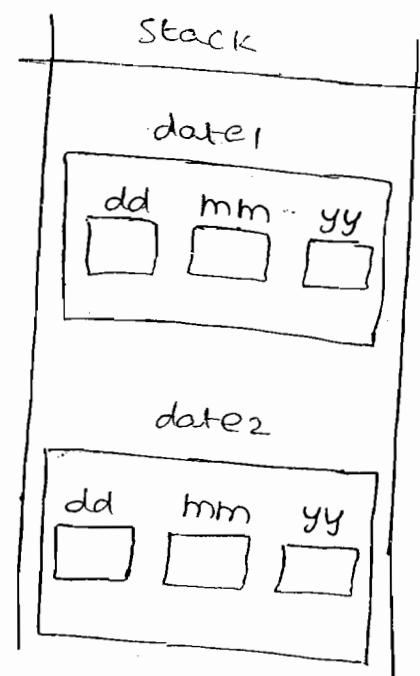
program :

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

struct date
{
    int dd, mm, yy,
};

int operator == (date d1, date d2)
{
    if (d1.dd == d2.dd && d1.mm == d2.mm && d1.yy == d2.yy)
        return 1,
    else
        return 0,
}

int main()
{
    date date1, date2,
    clrscr(),
    cout << "In Input date1 :",
    cin >> date1.dd >> date1.mm >> date1.yy,
    cout << "In Input date2 :",
    cin >> date2.dd >> date2.mm >> date2.yy,
    if (date1 == date2)
        cout << "In Dates are equal",
    else
        cout << "In Dates are not equal",
    return 0,
}
```



OUTPUT  
Input date1: 2 7 2010

They follow the syntax ~~the~~ rules of original operators

- \* Some operators can not be overloaded.
- \* unary operators overloaded by a member function takes no explicit arguments.
- \* binary operators overloaded by a member function takes one explicit argument.
- \* binary operators must explicitly return a value.
- \* They must not attempt to change their own arguments.

Program :

overloading + operator for concat two strings

```
#include <iostream.h>
```

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

```
#include <string.h>
```

struct String

{

char str[20];

}

String operator +(String s1, String s2)

{

String s3;

```
strcpy(s3.str, s1.str);
```

```
strcpy(s3.str, s2.str);
```

return s3;

}

```
int main()
{
    string str1, str2, str3;
    cin >> str1 >> str2;
    cout << "In Input String1:" << str1;
    cout << "In Input String2:" << str2;
    str3 = str1 + str2;
    cout << "In After concat" << str3;
    return 0;
}
```

Program:

overload == operator for comparing two strings.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

struct string
{
    char str[10];
};

int operator == (string s1, string s2)
{
    if(strncmp(s1.str, s2.str) == 0)
        return 1;
    else
        return 0;
}
```

```

int main()
{
    string str1, str2;
    cin >> str1 >> str2;
    cout << "In Input String1:";
    cout << "In Input String2:";
    if (str1 == str2)
        cout << "In Strings are equal";
    else
        cout << "In Strings are not equal";
    return 0;
}

```

- \* overloaded operator can be member of some class.
- \* overloaded member operator operates on objects.

Program :

```

#include <iostream.h>
#include <conio.h>
class date
{
    int dd, mm, yy;
public:
    void read_date();
}

cout << "In Input date:";
cin >> dd >> mm >> yy

```

```
int operator ==(date d)
{
    if(dd==d.dd && mm==d.mm && yy==d.yy)
        return 1;
    else
        return 0;
}

int main()
{
    date date1, date2,
        class9c;
    cout<<"In Input date1:";

    date1.read_date();
    cout<<"In Input date2:";

    date2.read_date();
    if(date1 == date2)
        cout<<"In Equal";
    else
        cout<<"In Not equal";
    return 0;
}
```

Program :

overloading + operator for adding two matrices

```
#include <iostream.h>
#include <conio.h>
class matrix
{
    int a[3][3];
public:
    void read_matrix()
    {
        cout << "Input elements of matrix";
        for (int i=0; i<3; i++)
            for (int j=0; j<3; j++)
                cin >> a[i][j];
    }
    void print_matrix()
    {
        cout << "Elements are:\n";
        for (int i=0; i<3; i++)
        {
            for (int j=0; j<3; j++)
                cout << a[i][j] << " ";
            cout << "\n";
        }
    }
}
```

matrix operator + (matrix m2)

{

matrix m3;

```

for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
        m3.a[i][j] = a[i][j] + m2.a[i][j];
    return m3;
}

```

```

int main ()
{
    matrix matrix1, matrix2, matrix3;
    class C;

    matrix1.read();
    matrix2.read();
    matrix3 = matrix1 + matrix2;
    matrix3.print();
    return 0;
}

```

**Output:** Input elements of matrix:

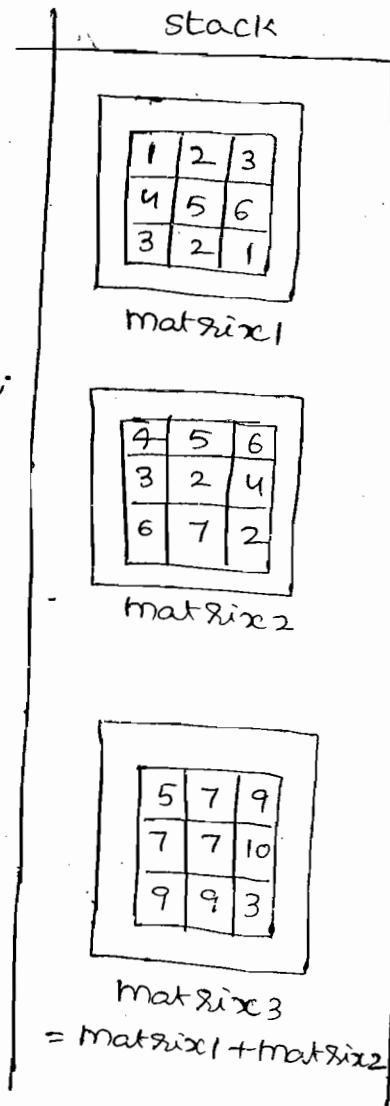
1	2	3
4	5	6
3	2	1

Input elements of matrix:

4	5	6
3	2	4
6	7	2

Elements are:

5	7	9
7	7	10



Program:

overloading unary ++, -- operators.

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

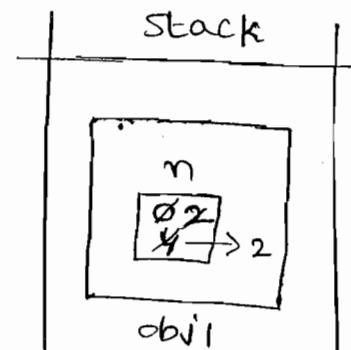
class A
{
    int n;
public:
    A()
    {
        n=0;
    }

    void operator ++( )
    {
        n=n+2;
    }

    void operator --( )
    {
        n=n-2;
    }

    void print( )
    {
        cout<<"n"<<n;
    }
};

int main( )
{
    A obj1;
    clrscr();
    obj1++;
    obj1.print();
    obj1++;
}
```



OUTPUT:

2  
4  
2

```
obj1.print();
obj1--;
obj1.print();
return 0;
}
```

overloaded with friend function :-

- \* operators can be overloaded using a friend keyword.
- \* unary operators overloaded by friend takes one reference argument.
- \* Binary operators overloaded by friend takes two reference arguments.
- \* certain operators cannot be overloaded using friend.
- \* The operators that cannot be overloaded using friend function.

Assignment operator =

Function call operator ()

Subscripting operator []

Class member access operator →

member functions can be used to overload them.

Syntax:

friend return-type operator operator-symbol (parameter);

return-type operator operator-symbol (parameter);

{

statements,

- \* The definition of friend operator is given within class or outside the class.

program

overloading of == operator for comparing two different objects.

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

class B;
class A
{
    int x, y;
public:
    void read-a();
};

cout << "In Input x, y values";
cin >> x >> y;
}

friend int operator == (A, B);
};

class B
{
    int p, q;
public:
    void read-b();
};

cout << "In Input p, q values";
cin >> p >> q;
}
```

```

friend int operator == (A, B);
};

int operator == (A o1, B o2)
{
    if (o1.x == o2.p && o1.y == o2.q)
        return 1;
    else
        return 0;
}

int main ()
{
    A obj1;
    B obj2;
    char c[1];
    obj1.read-a();
    obj2.read-b();
    if (obj1 == obj2) // == (obj1, obj2)
        cout << "In objects are equal ";
    else
        cout << "In objects are not equal ";
    return 0;
}

```

Output :

Input x, y values :

10  
20

Input p, q values :

10  
20

objects are equal,

- \* Insertion and Extraction operators operate on predefined data types does not operate on user defined data type.
- \* In order to operate on user defined types these operators are overloaded.

Insertion ( << ), Extraction ( >> )

Program :

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

class student
{
    int rno;
    char name[20];
public:
    friend void operator << (ostream &, student &),
    friend void operator >> (istream &, student &),
};

void operator << (ostream &out, student &s)
{
    out << s.rno << endl;
    out << s.name;
}

void operator >> (istream &in, student &s)
{
    in >> s.rno >> s.name;
}
```

```

int main ()
{
    Student stud1;
    classC();
    cout << "In Input student details : ";
    cin >> stud1;
    cout << "In student details : ";
    cout << stud1;
    return 0;
}

```

Output:

Input student details :

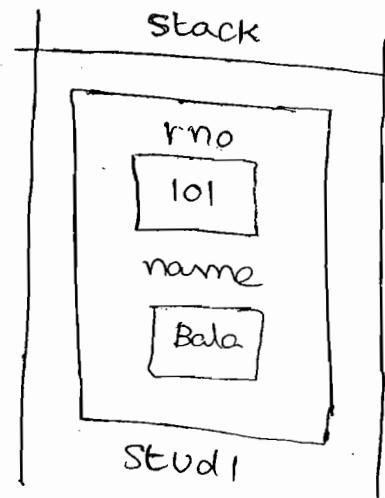
101

Bala

Student details :

101

Bala



# INHERITANCE

The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as base class and new ~~one~~ one is called as derived class.

A class can inherit properties from more than one class or from more than one level.

Advantages of Inheritance :-

~~~~~ ~~~ ~~~

1. Reusability.

2. Extensibility.

Defining Derived class :-

~~~~~ ~~~ ~~~

A derived class is defined by specifying its relationship with the base class.

The general form is:

class derived class : visibility mode base class

{                      ↳ Inheritance operator

members of derived class

}

The colon indicates that the derived class is derived from base class.

The visibility mode may be either private or public and is optional.

class A composition classB  
{

    {  
        Reusability obj.;

Inheritance → derived class

class C : A → Base class  
{

1. Reusability  
2. Extensibility.

Program : Example for composition

class book

{

public:

char bname[20];

char aname[20];

},

class publishers

{

float price;

book b;

public:

Void read\_details()

{

cout<<"Input book Name",

cin>>b.bname;

cout<<"Input author Name",

cin>>b.aname;

cout<<"Input price",

cin>>price;

}

Void print\_details()

{

cout<<"Book Name" << b.bname,

cout<<"Author Name" << b.aname;

cout<<"Price" << price;

}

,

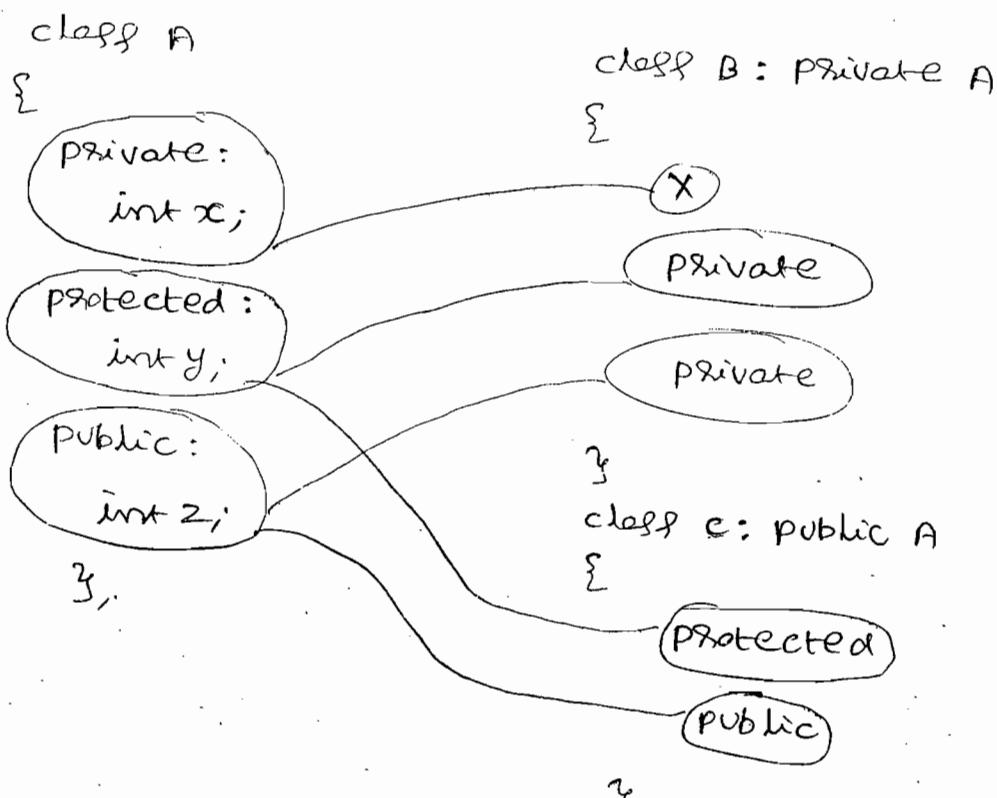
```

int main()
{
    publisher publisher1,
    publisher1.readDetails(),
    publisher1.printDetails(),
}

```

class derived-class : base-class  
 {  
 } → It means Inheritance operator  
 ↓  
 It means visibility/Access  
 of base class members within  
 derived class.

- \* default visibility mode is private
  - \* visibility modes are private, public, and protected.
- Example:



- \* The default visibility mode is private
- \* When the visibility mode is private, public members of the base class becomes the private members of the derived class.
- \* The public members of the base class are accessible by the member functions of the derived class only.
- \* They are not accessible to the derived class objects.

Note :

- \* Private members of a class not inherited but protected members of a class inherited.

Similarity :

Private and protected members cannot access the outside the class.

- \* If the visibility mode is public, public members of the base class becomes the public members of the derived class.
- \* They are accessible to the derived class objects.
- \* In both cases private members of the base class are not accessible to the derived class member functions.

Note :-

Size of object is sum of data members.

Eg:- int x, y, p, q

size = 8 bytes

x → 2 bytes

y → 2 bytes

p → 2 bytes

q → 2 bytes

Types of Inheritance :- C++ supports 5 types of Inheritance. They are:

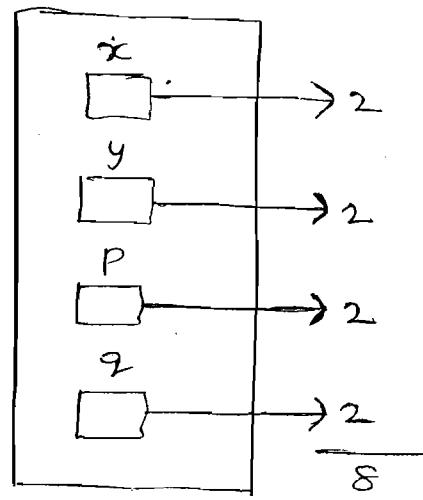
1. Single level Inheritance.
2. multi-level Inheritance.
3. multiple Inheritance.
4. Hierarchical Inheritance.
5. Hybrid Inheritance (or) multipath Inheritance

What happens when the object of derived class is created :-

```
class A
{
    int x, y;
};

class B : public A
{
    int p, q;
};

int main()
{
    B obj1;
    cout << sizeof(obj1);
}
```



- \* on creation of derived class object memory is allocated for data members of base class and derived class.
- \* If class does not have any data members the size is 1 byte

### Single level Inheritance:

A derived class with only one base class is called Single level Inheritance.

Ex:

class base

{

---

};

class derived : public base

{

---

};

Program : write a program for student details

```
#include <iostream.h>
```

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

```
class student
```

```
{
```

```
int rno;
```

```
char name[20];
```

```
public:
```

```
void •read - student();
```

```
cout << "In Input rollNo:";  
cin >> rno;  
cout << "In Input name:";  
cin >> name;  
}  
void printStudent()  
{  
    cout << "In RollNo" << rno;  
    cout << "In Name" << name;  
}  
};  
  
class marks : public student  
{  
    int sub1, sub2, sub3;  
public:  
    void readMarks()  
    {  
        cout << "In Input 3 subjects marks:";  
        cin >> sub1 >> sub2 >> sub3;  
    }  
    void findResult()  
    {  
        if (sub1 < 40 || sub2 < 40 || sub3 < 40)  
            cout << "In Result is fail";  
        else  
            cout << "In Result is pass";  
    }  
};
```

16  
int main ()

{

marks stud1;

clrscr();

stud1.read\_student();

stud1.read\_marks();

stud1.print\_student();

stud1.find\_result();

return 0;

}

Output :

Input rollNo: 37

Input name: Bala

Input 3 Subject marks: 90 95 85

RollNo: 37

Name: Bala

Result is pass

Input rollno : 50

Input name: Lalitha

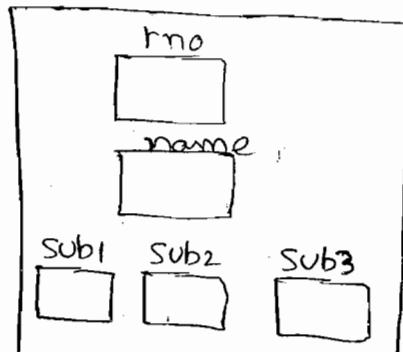
Input 3 Subject marks: 95 80 35

Roll No: 50

Name : Lalitha

Result is Fail

Stack



## Constructors in Inheritance :

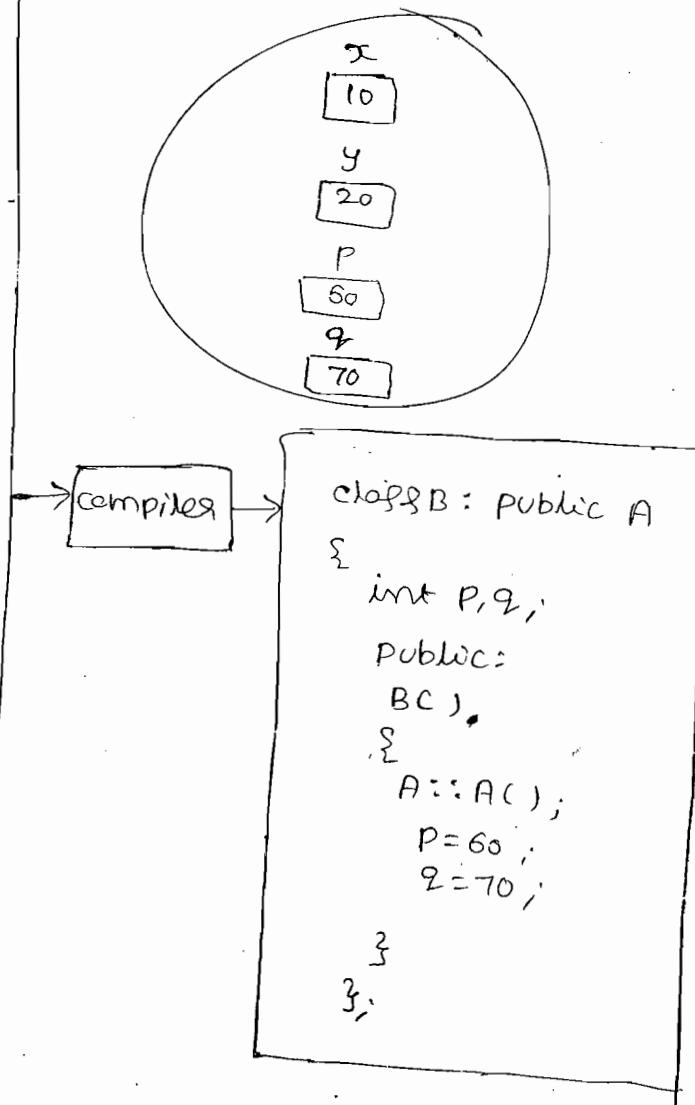
\* Default constructor of base class is bind with constructor available in derived class. This binding is done by compiler.

Ex:

```
class A
{
    int x, y;
public:
    A()
    {
        x=10;
        y=20;
    }
}

class B : public A
{
    int p, q;
public:
    B()
    {
        p=60;
        q=70;
    }
}

int main()
{
    B obj1;
}
```



Program :

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

class A
{
    int x, y;
public:
    A()
    {
        x=10;
        y=20;
    }
    void print_a()
    {
        cout<<"In x = "<<x;
        cout<<"In y = "<<y;
    }
};
```

```
class B : public A
```

```
{
    int p, q;
public:
    B()
    {
        p=60;
        q=70;
    }
};
```

```
void print_b()
```

```
{
    cout<<"In p = "<<p;
    cout<<"In q = "<<q;
}
```

```
};
```

Output :

x=10

y=20

p=60

q=70

```
int main()
{
    B obj1;
    obj1.print-a();
    obj1.print-b();
    return 0;
}
```

- \* only default constructor is bind implicitly, other constructors of class bind explicitly by user.

Syntax:

class-name:: constructor-name

- \* It is mandatory to write default constructor within base class when if any other constructor is define.

Program:

Explicit calling of base class constructor in derived class.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

class Student
{
    int rno;
    char name[20];
public:
    Student()
{
```

```
student (int r, char n[ ]) {
```

```
    rno = r;
```

```
    strcpy (name, n);
```

```
}
```

```
void print_student ()
```

```
{
```

```
    cout << "In Roll No: " << rno;
```

```
    cout << "In Name: " << name;
```

```
}
```

```
,
```

```
class marks : public student
```

```
{
```

```
    int sub1, sub2, sub3;
```

```
public:
```

```
marks (int r, char n[], int s1, int s2, int s3)
```

```
{
```

```
    student::student (r, n);
```

```
    sub1 = s1;
```

```
    sub2 = s2;
```

```
    sub3 = s3;
```

```
}
```

```
void print_marks ()
```

```
{
```

```
    cout << "In Subject1: " << sub1;
```

```
    cout << "In Subject2: " << sub2;
```

```
    cout << "In Subject3: " << sub3;
```

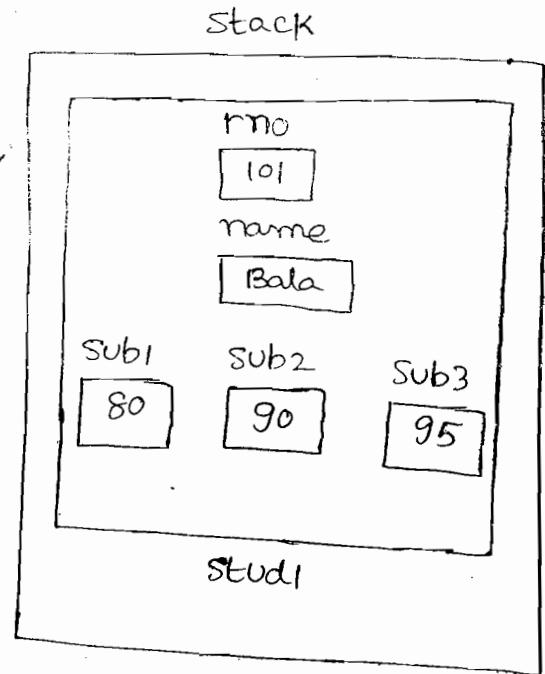
```
}
```

```
,
```

```

int main()
{
    marks stud1(101, "Bala", 80,
                90 95),
    class();
    Stud1.print_student();
    Stud1.print_marks();
    return 0;
}

```



output :-

Roll No: 101

Name: Bala

Subject1: 80

Subject2: 90

Subject3: 95

note :-

\* only one constructor of base class is called  
with in derived class.

marks(int r, char n[], int s1, int s2, int s3)

{ Student :: Student();

Student :: Student(r, n),

Sub1 = s1;

Sub2 = s2;

Sub3 = s3;

}

This is an Error

- \* binding of base class constructor with in derived class must be first statement.

marks ( int r, char n[], int s1, int s2, int s3)

{

Sub1 = s1;

Sub2 = s2;

Sub3 = s3;

student :: student(r, n),

}

This is an Error.

### Destructors in Inheritance :

- \* destructor of base class is bind implicitly with destructor available in derived class.
- \* Binding of base class destructor with in derived class is done as last statement.

Ex:

```
class A
{
public:
~A()
{
cout << "Inside destructor of class A";
}

class B: public A
{
public:
~B()
{
cout << "Inside destructor of class B";
}
}
```

```
class B: public A
{
public:
~B()
{
cout << "Inside
destructor of class B";
A:: ~A();
}

}
```

Program :

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

class A
{
public:
    ~A()
{
    cout<< "Inside destructor of class A";
}

class B: public A
{
public:
    ~B()
{
    cout<< "Inside destructor of class B";
}

int main()
{
    class A();
    B obj;
    return 0;
}
```

\* In the above program before deleting of object B  
destructor of B is called which calls the destructor  
of A class

\* Destructor can be called Explicitly using ~~del~~  
class.name:: destructor name

Ex: A:: ~A()

## Protected member:

A protected member is accessible by the member functions within its class and any class immediately derived from it.

It can not be accessed by the functions outside these two classes.

Ex:

```
class A
{
protected:
    int x, y;
private:
    int z;
};
```

```
class B : public A
{
public:
    void func()
    {
        cout << x << y;
        cout << z; → not Accepted
    }
};
```

Program:

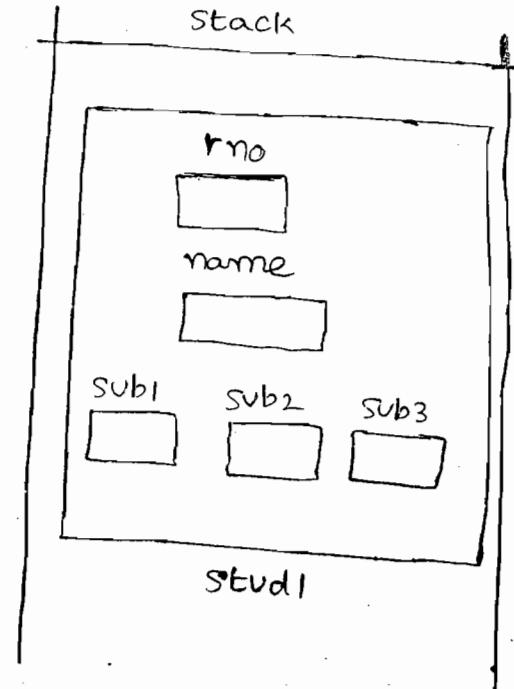
```
#include <iostream.h>
#include <conio.h>
class student
{
protected:
    int rno;
    char name[20];
public:
    void read_student()
    {
        cout << "Input rollno: ";
        cin >> rno;
        cout << "Input name: ";
        cin >> name;
    }
};
```

```

class marks : public student
{
    int sub1, sub2, sub3;
public:
    void read_marks()
    {
        cout << "In Input 3 Subject marks: ";
        cin >> sub1 >> sub2 >> sub3;
    }
    void find_result()
    {
        cout << "In RollNo: " << rno;
        cout << "In Name: " << name;
        if (sub1 < 40 || sub2 < 40 || sub3 < 40)
            cout << "In Result is Fail";
        else
            cout << "In Result is pass";
    }
};

int main()
{
    marks stud1;
    class();
    stud1.read_student();
    stud1.read_marks();
    stud1.find_result();
    return 0;
}

```



## Dynamic objects :

- \* Allocating memory for object during runtime.
  - \* Creating object in Heap area.
  - \* Dynamic memory allocation is done using new, delete operators.
1. Declare a pointer of type class.
  2. Allocate memory object using new operator.

Eg:

```
class A
{
    int x,y;
}
int main()
{
    A *P;           → Not object. It is a pointer
    P = new A;     which hold Address of
                    object.
}
```

Note :

constructor can be called Explicitly using  
new-operator →

Program :

```
#include <iostream.h>
```

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

```
class employee
```

```
{
```

```
    int empno;
```

```
    char name[20];
```

```
    float salary;
```

public:

```
void read_employee();
```

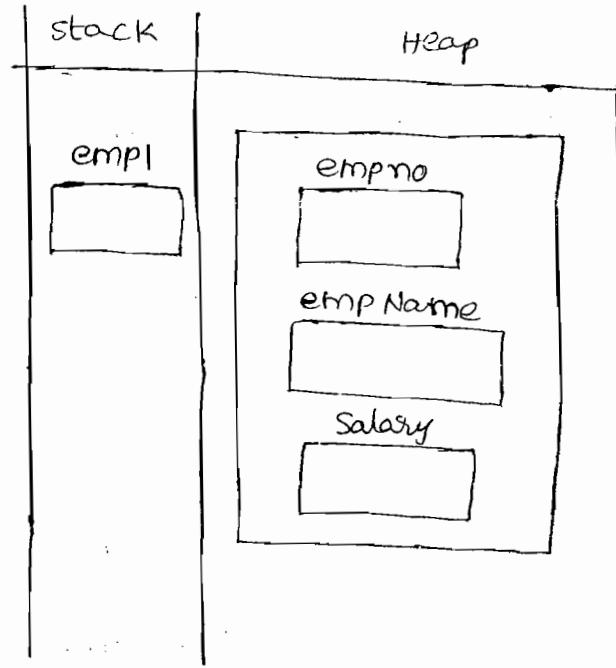
```

cout << "In Input employee no: ";
cin >> empno;
cout << "In Input employee Name: ";
cin >> name;
cout << "In Input employee Salary: ";
cin >> salary;
}

void print_employee()
{
    cout << "In Employee No: " << empno;
    cout << " In Employee Name: " << name;
    cout << " In Employee salary : " << salary;
}

int main()
{
    employee *empl;
    clrscr();
    empl = new employee();
    (*empl).read_employee();
    (*empl).print_employee();
    return 0;
}

```



Output:

```

Input employee no: 101
Input employee name : Bala
Input employee salary : 10,000

```

```

Employee No: 101
Employee Name: Bala
Employee Salary: 10,000

```

How to create dynamic Array of objects:

Allocating memory for more than one object of similar type.

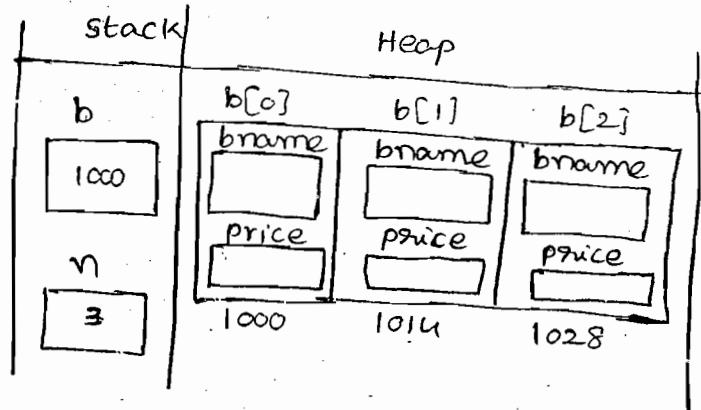
Program:

```
#include <iostream.h>
#include <conio.h>
class book
{
    char bname[10];
    float price;
public:
    void read_book();
    void print_book();
};

cout<<"In Input book name:";
cin>>bname;
cout<<" In Input price:";
cin>> price;
}

void print_book()
{
    cout<< endl << bname << "It" << price;
}

int main()
{
    book *b;
    int n,i;
    clrscr();
    cout<<"In Input How many books? ";
    cin>>n;
```



```
b = new book [n];  
for (i=0; i<n; i++)  
    b[i].read_book();  
for (i=0; i<n; i++)  
    b[i].print_book();  
return 0;  
}
```

Output :

Input how many books ? 3

Input book name : c

Input price : 300

Input book name : c++

Input price : 500

Input book name : Java

Input price : 700

c 300

c++ 500

Java 700

Program : creating <sup>dynamic</sup> derived class object

```
#include <iostream.h>
```

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

```
class A
```

```
{
```

```
public:
```

```
void function1()
```

```
{
```

```
cout << "Inside function1 of class A";
```

```
}
```

```
,
```

```

class B : public A
{
public:
    void function2();
}

cout << "Inside function2 of class B";
}

int main()
{
    B *obj;
    obj = new B;
    (*obj).function1();
    (*obj).function2();
    return 0;
}

```

member function overriding :

- \* Redefining of base class member function within derived class is called function overriding.
- \* In function overriding derived class function is defined with same signature of base class.
- \* Function overriding to allow to extend the functionality of base class with in derived class.

Ex:

```
class A
```

```
{
```

```
public:
```

```
void function1();
```

→ This is function overriding

```

        cout << "In Hello";
    }
}

class B : public A
{
public:
    void function1();
{
    cout << "Hello";
    cout << "In Bye";
}
};

int main()
{
    B obj1;
    class1;
    obj1.function1();
    return 0;
}

```

This is overridden function

- \* In function overriding the functionality of base class is hidden with in derived class.
- \* Base class function can't call outside the derived class but it is called with in derived class using the following syntax:

base-class:: function-name

Program :

```

#include <iostream.h>
#include <conio.h>
class student
{
    char name[20];
    int m1;

```

public:

void read ()

{

cout << "In Input rollno: ",

>> rno;

cout << "In Input name: ",

>> name;

}

void print ()

{

cout << "In Roll no: " << rno;

~~cout << "In Name: " << name;~~

}

};

class marks : public Student

{

int sub1, sub2, sub3;

public :

void read ()

{

Student :: read ();

cout << "In Input 3 subjects: ",

>> sub1 >> sub2 >> sub3;

}

void print ()

{

Student :: print ();

if (sub1 < 40 || sub2 < 40 || sub3 < 40)

cout << "In Result fail: ",

else

cout << "In Result pass ",

}

};

```
int main ()  
{  
    struct stud;  
    clas9 c;  
    stud.read();  
    stud.print();  
    return 0;  
}
```

### Static Binding :

In function overloading or operator overloading there will be more than function with the same name. The functions are invoked by matching arguments. The compiler selects the appropriate function for a particular call at the compilation time itself. This is called early binding or static binding.

### Dynamic Binding :

Selecting the appropriate member function while the program is running is known as runtime polymorphism. This process is known as late binding. It also known as dynamic binding as the selection of the function is done dynamically at runtime.

Dynamic binding requires use of pointers to objects.

The ~~late~~ feature of runtime polymorphism is the ability to refer to objects with out regard to their classes using a single pointer.

we can use a pointer to a base class to refer all the derived objects.

when we use the same function name in both base and derived classes a base class pointer executes the base class function only even when it is assigned with derived class address.

Ex:

```
class A
{
public:
    void fun1()
    {
        cout<< "fun1";
    }
};

class B : public A
{
public:
    void fun1()
    {
        cout<< "Derived fun1";
    }

    void fun2()
    {
        cout<< "Derived fun2";
    }
};
```

Virtual functions:

To invoke the derived class member function by pass class pointer when it is assigned with derived class address the base class member function has to be made virtual.

The keyword **virtual** has to be ~~preceded~~ to the

```
int main()
{
    B *child;
    child = new B;
    (*child). fun1();
    (*child). fun2();

    A *parent;
    parent = new A;
    (*parent). fun1();
    return 0;
}
```

Rules for virtual functions :-

- \* The virtual functions must be members of some class.
- \* They can not be static members.
- \* They are accessed by using object pointers.
- \* A virtual function can be a friend of another class.
- \* A virtual function in a base class must be defined even though it may not be used.
- \* The prototype of the base class virtual function and all the derived class functions must be identical.
- \* We can not have virtual constructors, but we can have virtual destructors.

Ex : Syntax :

virtual return-type function-name(parameters)

{

statements ;

}

Program :

```
#include <iostream.h>
```

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

```
class A
```

```
{
```

```
public:
```

```
virtual void function() {
```

```
{
```

```
cout << "Inside function of class A";
```

```
}
```

```
,
```

```
class B : public A
{
public:
    void function1()
    {
        cout << "In Inside function1 of class B";
    }
    void function2()
    {
        cout << "In Inside function2 of class B";
    }
};
```

```
int main()
{
    A *P;
    class C;
    P = new A;
    (*P).function1();
    delete P;
    P = new B;
    (*P).function1();
    (*P).function2(); // error
    return 0;
}
```

output :

② what is slicing?

(A) Slicing means that the data added by a subclass are discarded when an object of the subclass is passed or returned by value or from a function expecting a base class object.

Pure Virtual function :

- \* A function declared in base class and has no definition relative to the base class.
  - \* They are redefined in derived classes.
  - \* They are called as do-nothing functions.

Abstract base class:

- \* A class containing pure virtual functions is called abstract base class.
  - \* They can not be used to declare any objects.
  - \* It is purely for inheriting purpose only.
  - \* They are used to create a base pointer required to achieve runtime polymorphism.

Program :

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

class Shape

1

protected:

float dim<sub>1</sub>, dim<sub>2</sub>;

public:

void read ()

{

cout << "In Input 2dim",

cin >> dim1 >> dim2;

}

virtual float find-area () = 0;  
};  
 $\underline{(\text{? } \underline{\underline{d}} = \underline{\underline{e}})}$

class triangle : public shape

{

public:

float find-area ()

{

return 0.5 \* dim1 \* dim2;

};  
 $\underline{\underline{p}}$

class rectangle : public shape

{

public:

float find-area ()

{

return dim1 \* dim2;

};

int main()

{

triangle triangle1;

class();

triangle1.read();

float a1 = triangle1.find-area();

cout << "Area of triangle is " << a1;

```
rectangle rectangle1;
rectangle1.read();
float a2 = rectangle1.findArea();
cout << "Area of rectangle is " << a2;
return 0;
}
```

Another way to write main function of above prog.  
This is Runtime polymorphism:

```
int main()
{
    shape *shape1;
    shape1 = new shape();
    (*shape1).read();
    float a1 = (*shape1).findArea();
    delete shape1;
    shape1 = new rectangle();
    (*shape1).read();
    (*shape1).findArea();
}
```

- \* Pure virtual function ~~definition~~ definition must be given derived class.
- \* If not derived class becomes abstract class.

program:

```
#include <iostream.h>
```

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

```
class employee
```

```
{ protected:
```

```
    int empno;
```

```
    char name[10];
```

```
public:
```

```
    virtual void read() = 0;
```

```
    virtual void print() = 0;
```

```
}
```

```
class SalariedEmployee : public employee
```

```
{
```

```
    float salary;
```

```
public:
```

```
    void read();
```

```
{
```

```
    cout << "In Input employee no: ",
```

```
    cin >> empno;
```

```
    cout << "In Input employee name: ",
```

```
    cin >> name;
```

```
    cout << "In Input salary",
```

```
    cin >> salary;
```

```
}
```

```
    void print();
```

```
{
```

```
    cout << "In Employee No: " << empno;
```

```
    cout << "In Employee Name: " << name,
```

```
    cout << "In Salary" << salary;
```

```
}
```

class daily-worker : public employee

{

float wage;

public:

void read();

{

cout << "In Input employee no: ",

cin >> empno;

cout << "In Input employee name: ",

cin >> name;

cout << "In Input daily wage: ",

cin >> wage;

}

void print();

{

cout << "In Employee No: " << empno;

cout << "In Employee Name: " << name;

cout << "In Employee wage: " << wage;

}

,

int main();

{

employee \*emp;

desc();

emp = new SalariedEmployee;

(\*emp).read();

(\*emp).print();

delete emp;

```
emp = new daily_worker;  
(*emp).read();  
(*emp).print();  
return 0;  
}
```

virtual class :-

You might want to make a class virtual if it is a base class that has been passed to more than one derived class, as might happen with multiple inheritance.

A base can't be specified more than once in a derived class.

```
class B { ... };
```

```
class D : B, B { ... }; // Invalid
```

However, a base class can be indirectly passed to the derived class more than one

```
class x : public B { ... }
```

```
class y : public B { ... }
```

```
class z : public x, public y { ... }
```

In this case, each object of class z will have two subobjects of class B.

If this causes problems, you can add keyword "virtual" to a base class specified

Program: // Hybrid inheritance

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

class A
{
public:
    void fun1()
    {
        cout << "In Inside function1 of class A";
    }
};

class B: virtual public A
{
public:
    void fun2()
    {
        cout << "In Inside function2 of class B";
    }
};

class C: public virtual A
{
public:
    void fun3()
    {
        cout << "In Inside function3 of class C";
    }
};
```

```

class D: public B, public C
{
public:
    void fun4()
    {
        cout << "Inside function4 of class D";
    }
};

int main()
{
    D obj1;
    classA();
    obj1.fun1();
    obj1.fun2();
    obj1.fun3();
    obj1.fun4();
    return 0;
}

```

---

Virtual destructor :-

- \* Virtual destructor is used to implement dynamic binding.

Ex:

```

class A
{
public:
    virtual ~A()
    {
        cout << "Destructor of class A";
    }
}
```

```
class B: public A
{
public:
    ~B()
{
    cout << "Destructor of class B";
}
};

int main()
{
    A *p;
    p = new A;
    delete p;
    p = new B;
    delete p;
    return 0;
}
```

Note :-

\* There is no virtual constructor.

\* Compiler binds or invoke members using pointer type. In order to invoke based on object type we declared base class members as virtual.

## multiple Inheritance :

If a class is derived from more than one base class it is called multiple Inheritance.

Ex :

```
class base1
```

```
{
```

```
};
```

```
class base2
```

```
{
```

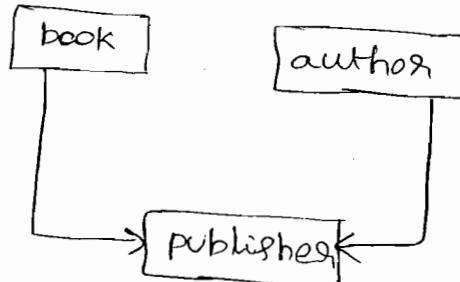
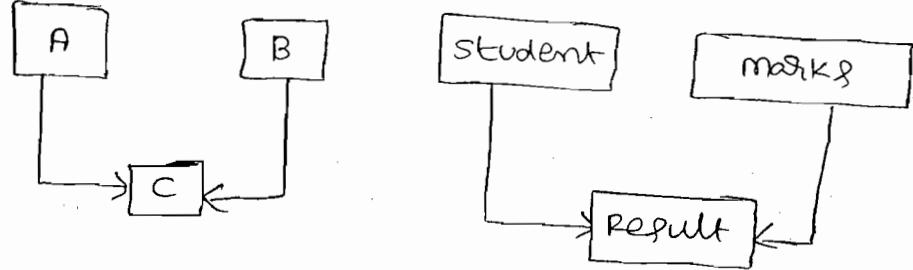
```
};
```

```
class derived : public base1, public base2
```

```
{
```

```
};
```

Examples :



## Program :

// multiple Inheritance

```
#include <iostream.h>
```

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

```
class Student
```

```
{
```

```
    int rno;
```

```
    char name[20];
```

```
public:
```

```
    void read_student();
```

```
{
```

```
    cout << "In Input roll no";
```

```
    cin >> rno;
```

```
    cout << "In Input name";
```

```
    cin >> name;
```

```
}
```

```
    void print_student();
```

```
{
```

```
    cout << "In Roll No: " << rno;
```

```
    cout << "In Name: " << name;
```

```
}
```

```
};
```

```
class Marks
```

```
{
```

```
protected:
```

```
    int sub1, sub2, sub3;
```

```
public:
```

```
    void read_marks();
```

```
{
```

```
    cout << "In Input 3 subject marks";
```

```
    cin >> sub1 >> sub2 >> sub3;
```

```
}
```

```

void print-marks()
{
    cout << "In Subject1:" << sub1;
    cout << "In Subject2:" << sub2;
    cout << "In Subject3:" << sub3;
}

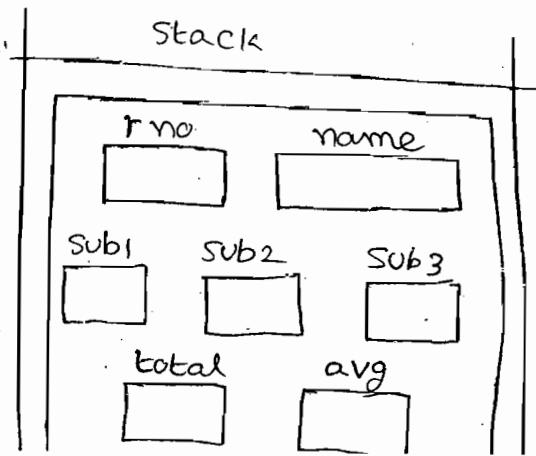
class Result : public student, public marks
{
    int total;
    float Avg;

public:
    void find-result()
    {
        total = sub1 + sub2 + sub3;
        Avg = total / 3.0;

        cout << "In Total:" << total;
        cout << "In Avg :" << Avg;

        if (sub1 < 40 || sub2 < 40 || sub3 < 40)
            cout << "In Result Fail";
        else
            cout << "In Result Pass";
    }
}

```



```

int main ()
{
    result stud1;
    class();
    stud1.read - student();
    stud1.read - marks();
    stud1.print - student();
    stud1.print - marks();
    stud1.find - result();
    return 0;
}

```

Hierarchical inheritance:

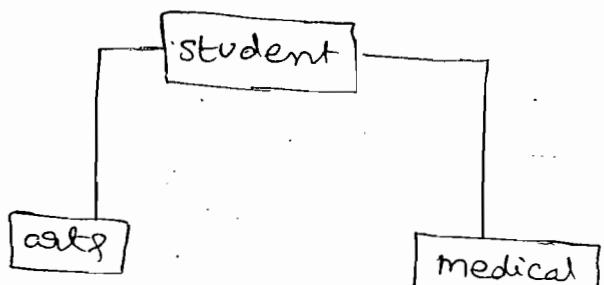
If class is inherited by more than one class it is called hierarchical inheritance.

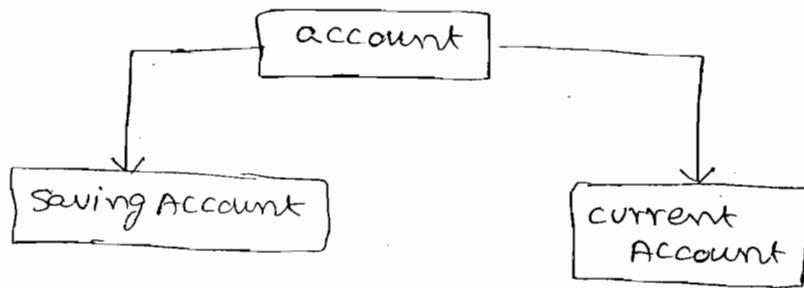
Ex:  
class student
{
 ...
}

class artl : public student
{
 ...
}

class medical : public student
{
 ...
}

Example:





Program :

```

#include <iostream.h>
#include <conio.h>

class account
{
protected:
    int accno;
    char cname[20];
    float balance;
public:
    virtual void read_account() = 0;
    virtual void print_account() = 0;
};

class savingaccount : public account
{
public:
    void read_account()
    {
        cout << "Input account-no: ";
        cin >> accno;
        cout << "Input customer name: ";
        cin >> cname;
    }
};

```

```
cout << "In Input balance : ";
cin >> balance;
}

void print - account ( )
{
    cout << "In Account No : " << accno;
    cout << "In Customer Name : " << cname;
    if (balance < 1000)
        cout << "In Invalid balance " << balance;
    else
        cout << "In balance " << balance;
}
};
```

```
class currentaccount : public account
{
```

```
public :
```

```
void read - account ( )
{
```

```
    cout << " In Input account no : ";
    cin >> accno;
    cout << " In Input customer name : ";
    cin >> cname;
    cout << " In Input balance : ";
    cin >> balance;
}
```

```
void print - account ( )
{
```

```
    cout << " In Account No : " << accno;
```

```
cout << "In customers Name: " << name;
```

```
if (balance < 2000)
```

```
cout << "In Invalid balance " << balance,  
else
```

```
cout << "In balance " << balance;
```

```
}
```

```
};
```

```
int main ()
```

```
{
```

```
account *acc,
```

```
classA(),
```

```
acc = new Saving account;
```

```
(*acc). read - account ();
```

```
(*acc). print - account ();
```

```
delete acc;
```

```
acc = new current account;
```

```
(*acc). read - account ();
```

```
(*acc). print - account ();
```

```
return 0;
```

```
}
```

Hybrid inheritance :

If there are two or more types of inheritances to design a program it is called as hybrid inheritance.

Ex :

class Student

{

};

class Test : public Student

{

};

class Sports : public Student

{

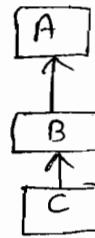
};

class Result : public Test, public Sports,

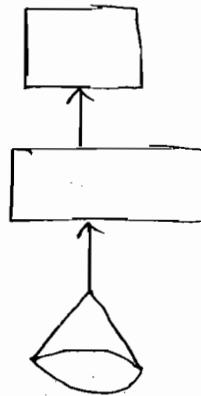
This is also called as multi path inheritance  
as the class Student inherited via Test and also  
via Sports.

multi-level Inheritance :

when the derived class is inherited from the other derived class, it is called multi-level inheritance. The typical block diagram of multi-level inheritance like is like this



Ex:



The general format of multi-level inheritance will be like this.

class A

{

====

};

class B : public A

{

====

};

class C : public B

{

====

};

Program :

The following program demonstrates The multi level Inheritance

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

class Square
{
protected :
    int l;
public :
    void Accept1()
    {
        cout<< "Enter length ";
        cin>> l;
    }
    void area1()
    {
        cout<< l*l;
    }
};
```

class Rectangle : public square

{

protected :
 int b;

public :

```
void Accept2()
{
```

Accept1(),
cout<< " Enter breadth ",

cin>> b;

}

Void Area 2 c,

{

cout << l \* b;

}

};

class Pyramid : public Rectangle

{

private :

int h;

public :

Void Accept 3 c,

{

Accept 2 c ,

,

cout << "Enter height ",

cin >> h;

}

Void area 3 c,

{

cout <<  $\frac{1}{3} \times b \times h$ ;

}

};

Void main c,

{

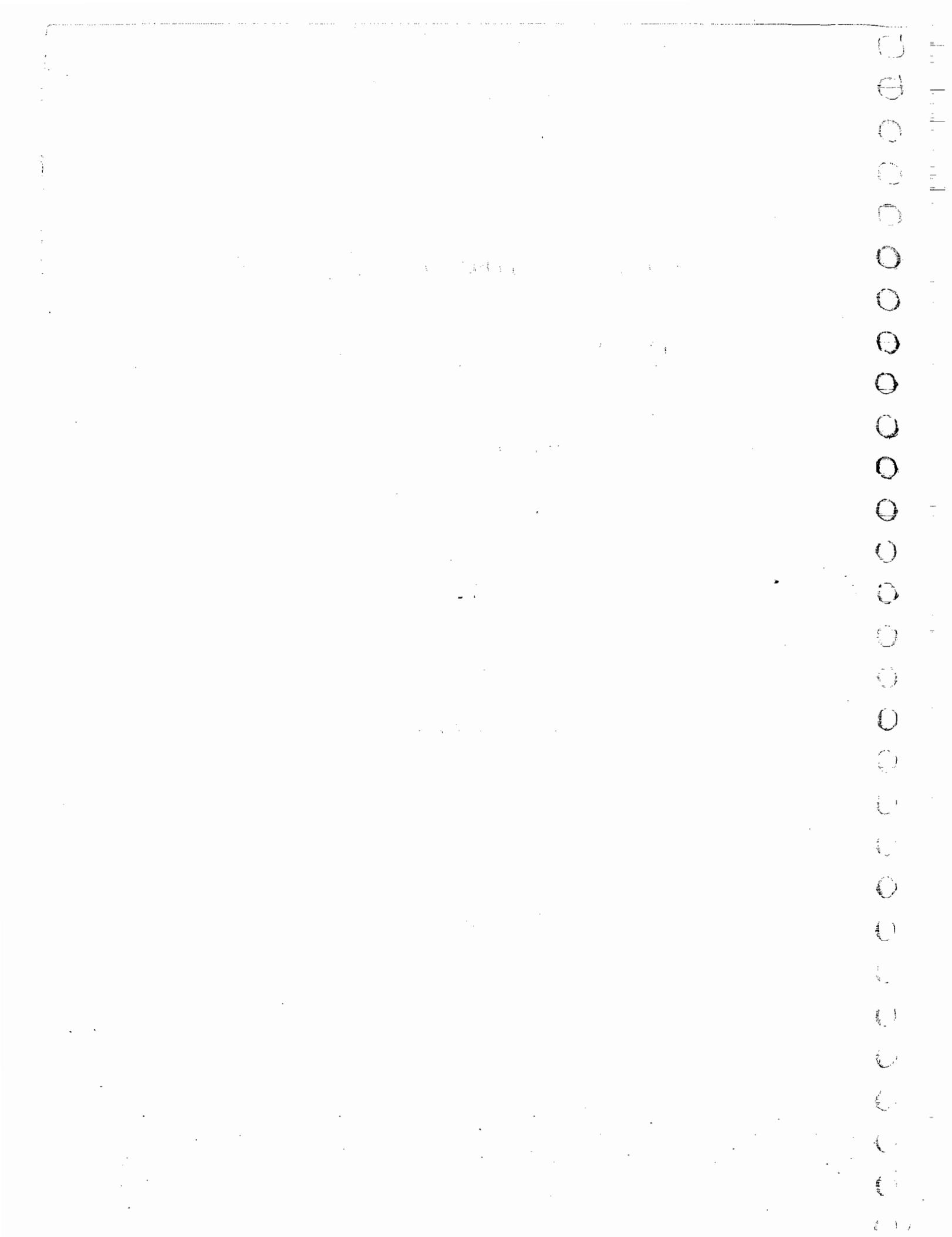
Pyramid obj;

obj. Accept pyramid c ,

obj. area c ,

getch c ,

}



## Templates or Generics :

- \* Templates enable us to create generic classes and functions
- \* A template can be used to create a family of classes or functions.
- \* In a generic class or function, the type of data upon which the function or class operates is specified as a parameter.
- \* Templates construct a family of related functions or classes.
- \* Templates are also called "generics" or "parametrized types", enable you to construct a family of related functions or classes.
- \* Use templates when you need to write lots of nearly identical things.
- \* Templates are classified two types
  1. Function templates.
  2. Class templates.

### Syntax :

template < class name, class name, ... >

Ex:- template < class T1 >



This template receive  
one datatype

Void function( T1 x, T2 y )  
{ }

template < class T1, class T2 >



This template receive  
two datatype as  
arguments.

what is generic function and function template:

- \* Defines a general set of operations that will be applied to various types of data
- \* A single general procedure can be applied to a wide range of data.
- \* Defines the nature of the algorithm independent of any data.
- \* compiler automatically generates the correct code for the type of data that is used at the time of compilation.

The general form of template function is

template < class TType >

```
return type funname (args)
{
    // Statements
}
```

TType is a place holder name for a data type used by the function. This name may be used within a function definition. The compiler automatically replaces it with an actual data type when it creates a specific version of the function.

Ex :

```
template < class T >
T max (T x, T y)
{
    if (x > y)
        return x;
    else
        return y;
```

```

int main()
{
    int a = max(10, 20);
    float b = max(1.5, 1.0);
    cout << a;
    cout << b;
    return 0;
}

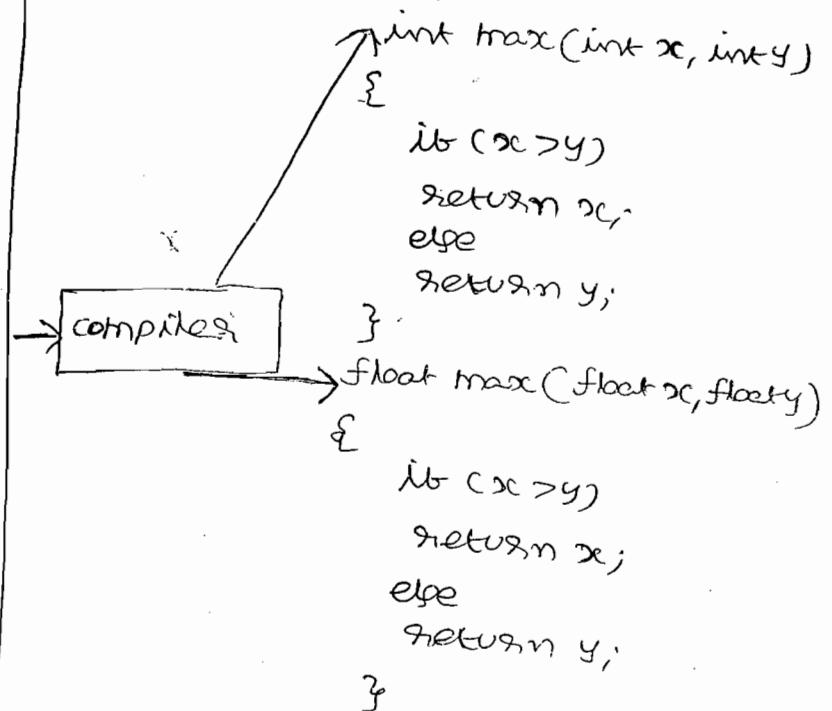
```

```

template <class T>
T max(T x, T y)
{
    if (x > y)
        return x;
    else
        return y;
}

int main()
{
    int a = max(10, 20);
    float b = max(1.5, 1.0);
    cout << a;
    cout << b;
}

```



program:

```

#include <iostream.h>
#include <conio.h>

template < class T1, class T2 >
void function1(T1 x, T2 y)
{
    cout << "In x = " << x,

```

```

int main ()
{
    classA();
    function1(10, 20);
    function1(1.5, 20);
    return 0;
}

```

OUTPUT :

x=10

y=20

x=1.5

y=20

overloading function templates or generic function :-

```

#include <iostream.h>
#include <conio.h>
template <class T>
T max(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}

```

```

template <class T>
T max(T a, T b, T c)
{
    if (a > b && a > c)
        return a;
    else
        if (b > c)
            return b;
        else
            return c;
}

```

```

int main()
{
    class();
    int p = max(10, 20);
    float q = max(1.5, 1.2);
    int m = max(20, 40, 30);
    float n = max(1.7, 1.5, 1.2);
    cout << "In" << p;
    cout << "In" << q;
    cout << "In" << m;
    cout << "In" << n;
    return 0;
}

```

Output :

20  
1.5  
40  
1.7

Generic class :

- \* Generic classes are useful when a class uses logic that can be generalized.
- \* A class that defines all algorithms can be created.
- \* Actual type of data being manipulated will be specified as a parameter when objects are created.
- \* For example the same algorithms that maintain a queue of integers will also work for a queue of floats.

The general form of a generic class declaration

```

Template <class Ttype>
class class-name
{

```

-----

Type name is a placeholder type name, which will be specified when a class is instantiated. we can define more than one generic data type.

General form of defining an object of generic class

class-name <type> obj;

type is the type name of the data that the class will be operating upon.

Q) Differentiate between a template class and class template

A) Template class:

A generic definition or a parameterized class not instantiated until the client provides the needed information.

class template :

A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed.

Ex:

template <class T>

class A

{

    T x, y;

public:

    Void print (,

{

        cout << x << y;

}

```

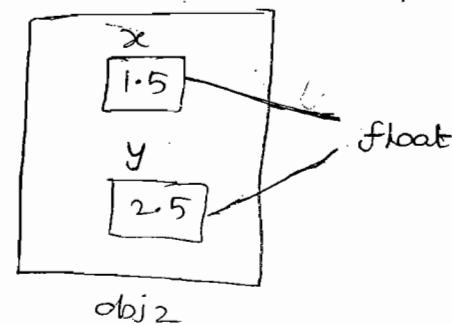
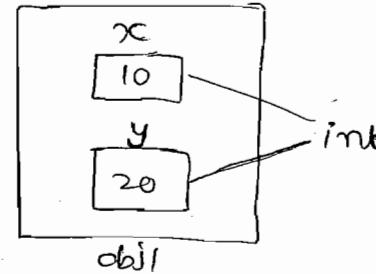
5
6 void set(T x, T y)
7 {
8     (*this).x = x;
9     (*this).y = y;
10 }
11

```

```

12 int main()
13 {
14     A<int> obj1;
15     A<float> obj2;
16     obj1.set(10, 20);
17     obj2.set(1.5, 2.5);
18     obj1.print();
19     obj2.print();
20     return 0;
21 }

```



Program:

```

# include <iostream.h>
# include <conio.h>
template <class T>
class vector
{
    T a[10];
    int size;
public:
    vector (int s)
    {
        size = s;
    }
}

```

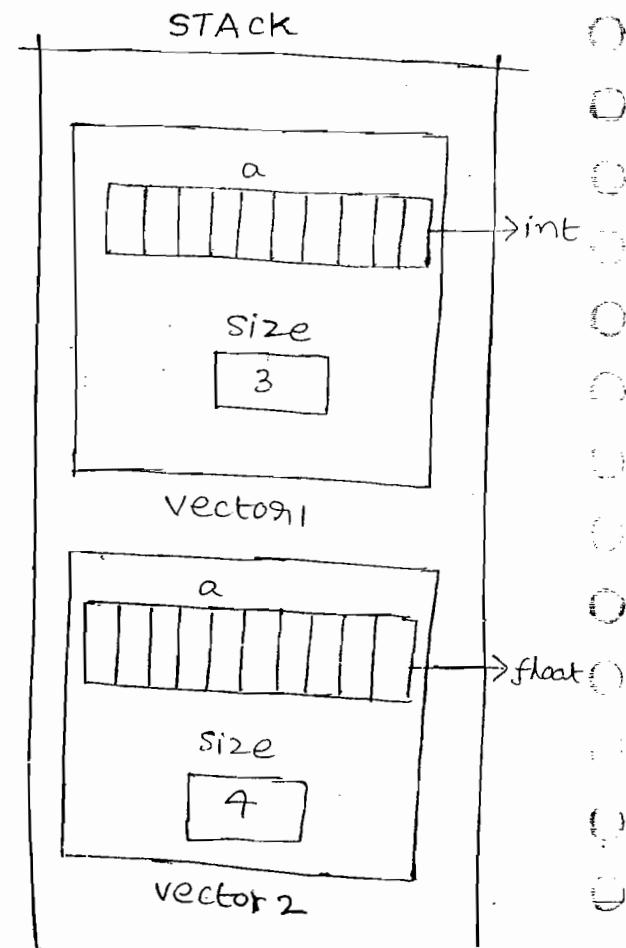
```

void read-elements() {
    cout << "In Input Elements of vector:";
    for (int i=0; i<size; i++)
        cin >> a[i];
}

void print-elements() {
    cout << "Elements are: ";
    for (int i=0; i<size; i++)
        cout << a[i];
}

int main() {
    vector<int> vector1(3);
    vector<float> vector2(4);
    clrscr();
    vector1.read-elements();
    vector1.print-elements();
    vector2.read-elements();
    vector2.print-elements();
    return 0;
}

```



Example of generic function :

```
template <class T>
void print (int x, T y)
{
    cout << "In" << x;
    cout << "In" << y;
}
int main()
{
    print (10, 20);
    print (10, 1.5);
    return 0;
}
```

Program :

A class can have multiple datatypes.

```
#include <iostream.h>
#include <conio.h>
template <class T1, class T2>
class A
{
    T1 x;
    T2 y;
public:
    void read ()
    {
        cout << "Input x,y values : ";
        cin >> x >> y;
    }
}
```

```
Void read ( )
```

```
{
```

```
cout << "In Input x, y values : ";
```

```
cin >> x >> y;
```

```
}
```

```
Void print ( )
```

```
{
```

```
cout << "In " << x;
```

```
cout << "In " << y;
```

```
}
```

```
,
```

```
int main ( )
```

```
{
```

```
A < int, float > obj1;
```

```
A < float, float > obj2;
```

```
clrscr ( );
```

```
cout << "In Size of object1 " << sizeof (obj1);
```

```
cout << "In Size of object2 " << sizeof (obj2);
```

```
obj1.read ( );
```

```
obj1.print ( );
```

```
obj2.read ( );
```

```
obj2.print ( );
```

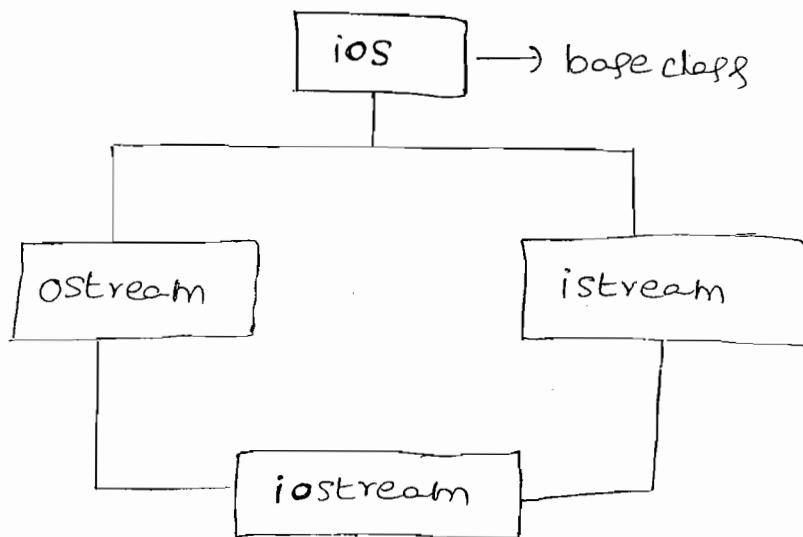
```
return 0;
```

```
}
```

→

## Streams and Files

- \* C++ uses the concept of Stream and Stream classes to implement I/O operations with console and disk files.
- \* The input functions are defined in `istream` class.
- \* The output functions are defined in `ostream` class.
- \* The `iostream` class inherits the properties of `istream` and `ostream` classes.



get() :-  
~~~~~

- \* member function of `istream` class.
- \* fetches a single character including a blank space, tab and new line characters. `get()` can be used in two ways.

`cin.get(ch);` // get a character from keyboard and assign it to ch.

`ch = cin.get();` also can be used.

program:

~~~~~

write a program to read sentence and count no. of characters, words, and lines

```
#include <iostream.h>
#include <conio.h>
int main()
{
    char ch;
    int wc=1, lc=1, cc=0;
    clrscr();
    while ((ch = cin.get()) != '*')
    {
        cc++;
        if (ch == ' ')
            wc++;
        else
            if (ch == '\n')
            {
                wc++;
                lc++;
            }
        cout << "No of characters: " << cc;
        cout << "No of words: " << wc;
        cout << "No of lines: " << lc;
        return 0;
    }
}
```

OUTPUT  
~~~~~  
Hello Bala\*

No of characters:	9
No of words:	2
No of lines:	1

PUT ( ) :-

~~~~~

member function of ostream class

cout.put(ch); // displays the value of the variable ch

It can also be used to output a line of text, character by character.

Ex :-

cout.put(65); → A

cout.put('B'); → B

getline ( ) :-

member function of istream class.

We can read the text more efficiently using line oriented getline ( ).

cin.getline (line, size);

Reads character input into the variable line

The reading is terminated as soon as either 'n' is encountered or size characters are read.

Program :-

~~~~~

```
#include <iostream.h>
```

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

```
int main()
```

```
{
```

```
char name[20],
```

```
clrscr();
```

```
cout<<"\n Input name:",
```

```
cin>>name; // cin.getline(name, 20);
```

```
cout<<"\n Name" << name,
```

```
return 0;
```

[  
    out put  
    ~~~~~  
    Input Name:  
    Rama Rao  
    Name: Rama Rao]

write() :-

member function of ostream class.

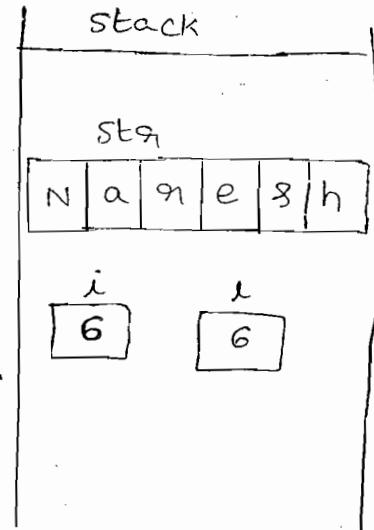
write() function displays the entire line.

cout.write(line, size);

line represents the ~~the~~ name of the string to be displayed. size indicates the number of characters to display. It does not stop displaying the characters when null character is encountered.

Program:

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
int main()
{
    char str[10];
    int i, l;
    clrscr();
    cout << "Input Any String: ";
    cin >> str;
    l = strlen(str);
    for (i = 1; i <= l; i++)
    {
        cout.write(str, i);
        cout << "\n";
    }
    return 0;
}
```

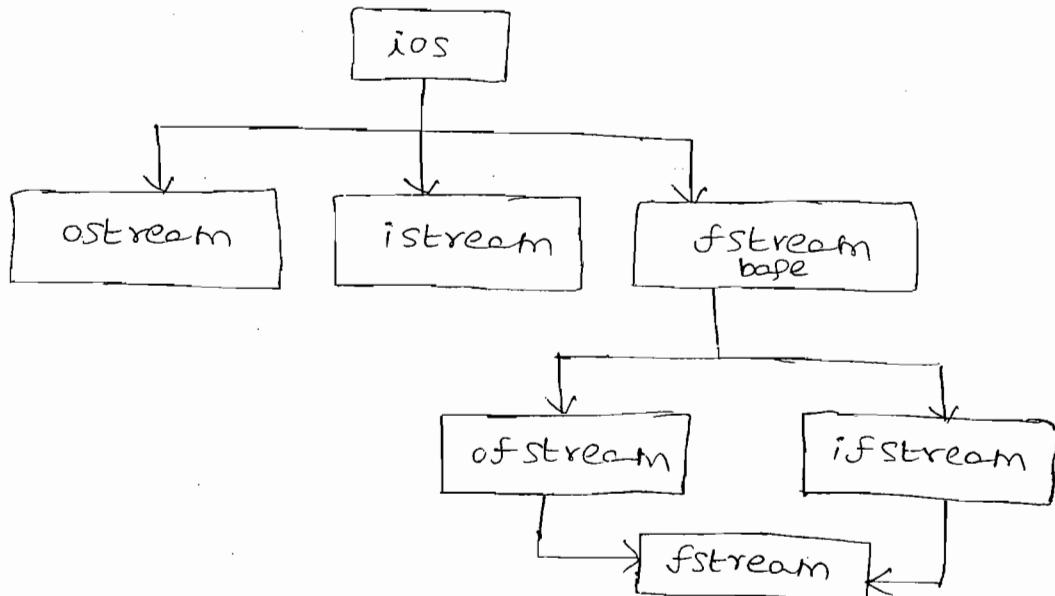


Output  
~~~~~  
Input Any String: Naresh

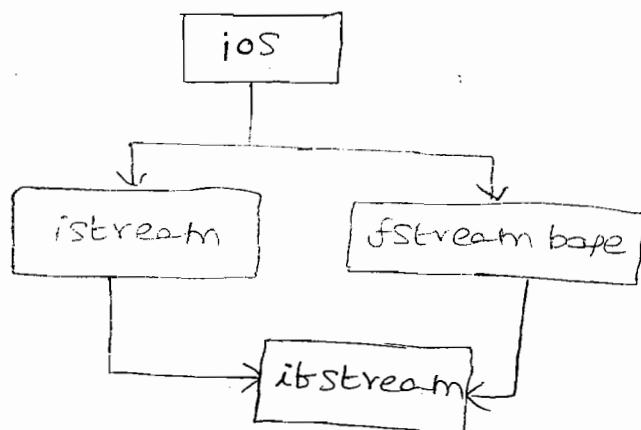
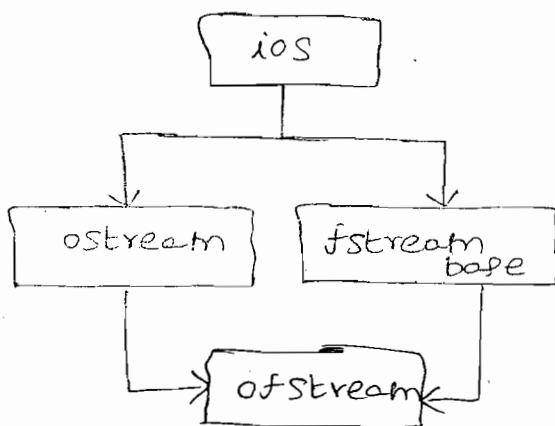
N  
Na  
Nan  
Name  
Names  
Naresh

## Files :

- \* File is a named memory location on external storage device.
- \* File is collection of data or information.
- \* File is used to save the data permanently.
- \* A file is a collection of related data stored in a particular area on the disk.
- \* C++ contains a set of classes for file handling.
- \* They are istream, ostream, fstream.
- \* These are declared in fstream.h
- \* istream is nothing but a input file stream.
- \* ostream is nothing but a output file stream.



- \* ofstream is a derived class of fstream base and ostream classes.
- \* ifstream is a derived class of fstreambase and istream classes.



- \* `ios` and `fstream base` are called abstract classes.

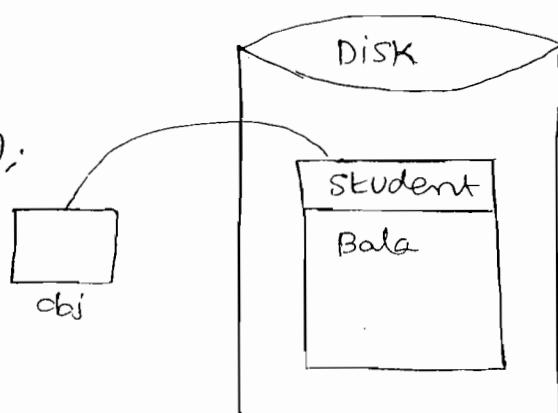
ofstream:

- \* output file stream class, which provides operations (or) functions to perform output (writing) operations on file/disk.
- \* `ofstream` creates new file for writing data.
- \* `open` is a member function of `ofstream` class, which open the file in output mode for writing data.

`ofstream obj;`

`obj.open ("Student");`

`obj << "Bala";`



program:

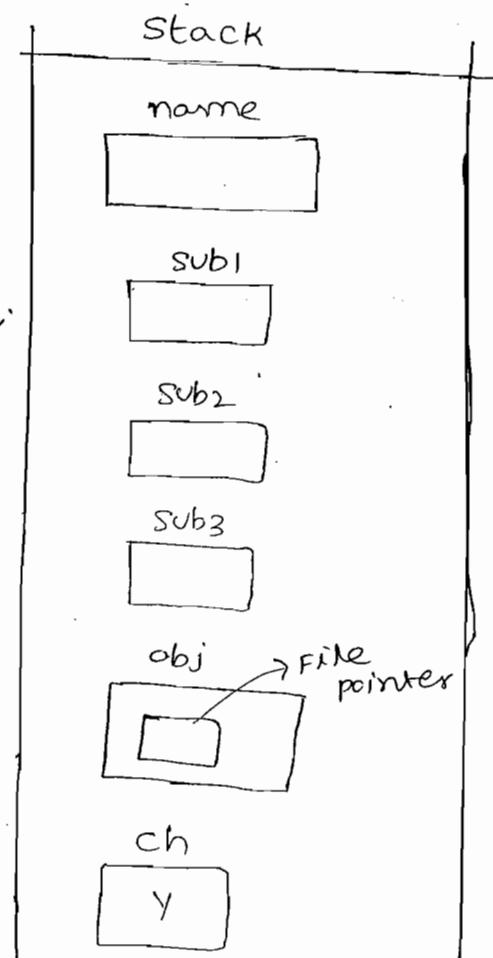
```
#include <iostream.h>
#include <iostream.h>
#include <conio.h>
int main()
{
    char name[20];
    int sub1, sub2, sub3;
    ofstream obj;
    char ch='y';
    obj.open("marks");
    clrscr();
    while(ch != 'n')
    {
        cout << "Input name: ";
        cin >> name;
        cout << "Input 3 subject marks: ";
        cin >> sub1 >> sub2 >> sub3;
        obj << name << endl;
        obj << sub1 << endl << sub2 << endl <<
            sub3 << endl;
        cout << "Add Another student? ";
        cin >> ch;
    }
    return 0;
}
```

[ Go to file → Dos shell ]  
[ C:\TCC> type marks ]

To see output:

To click on file to select Dos shell

C:\TCC> type marks



### ifstream :

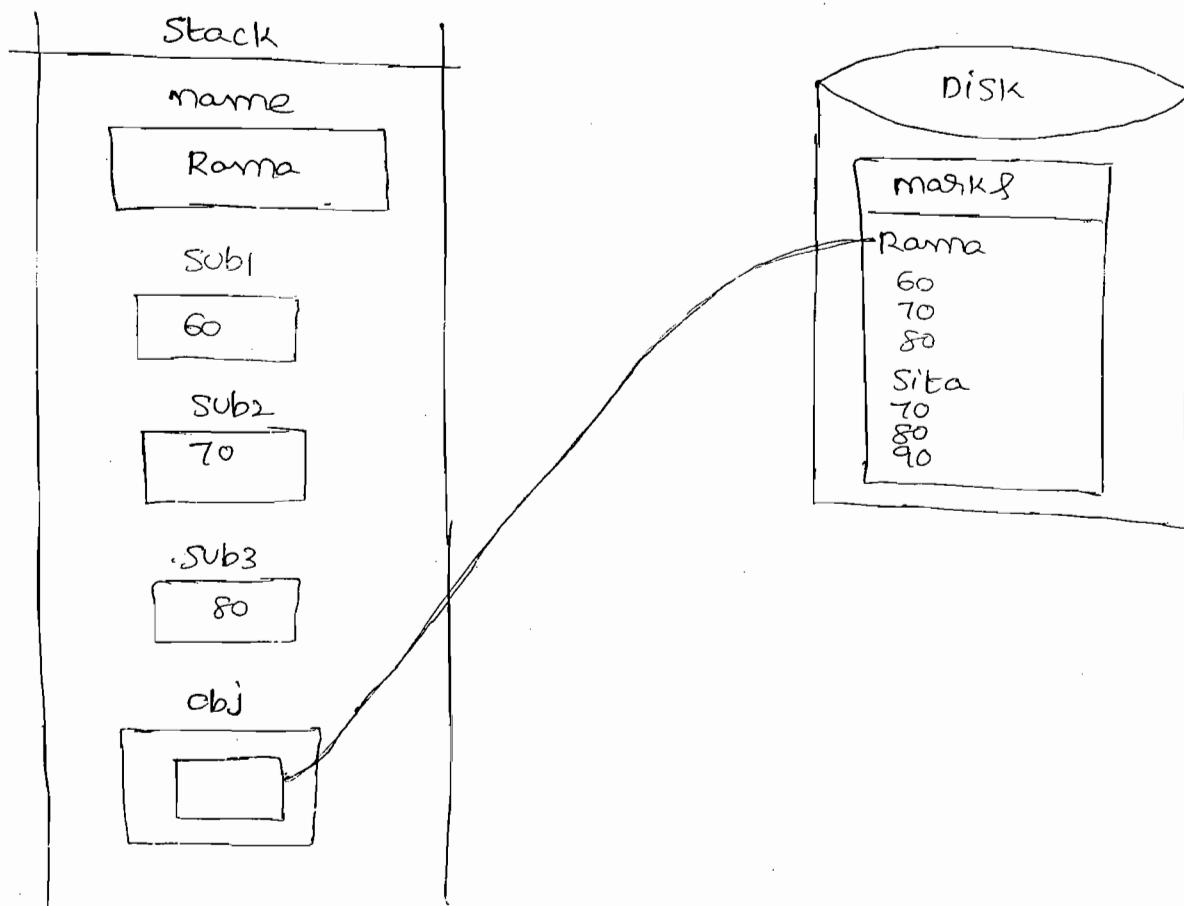
- \* It is called as input file stream, which provides member functions to read data from existing file.
- \* ifstream opens the file in input mode (reading).

### open( ) :

which opens the file in input mode.

`open ("file name")`

If file does not exist, it returns null



Program :

```
#include <iostream.h>
#include <iostream.h>
#include <conio.h>
int main()
{
    char name[20];
    int sub1, sub2, sub3;
    ifstream obj;
    obj.open ("Mark8");
    clrscr();
    while (1)
    {
        obj >> name;
        obj >> sub1 >> sub2 >> sub3;
        if (obj.eof())
            break;
        cout << "\n" << name << " " << sub1 << " " <<
            sub2 << " " << sub3;
        if (sub1 < 0 || sub2 < 0 || sub3 < 0)
            cout << "Fail";
        else
            cout << "Pass";
    }
    return 0;
}
```

Reading And writing objects :

writing object with in file :

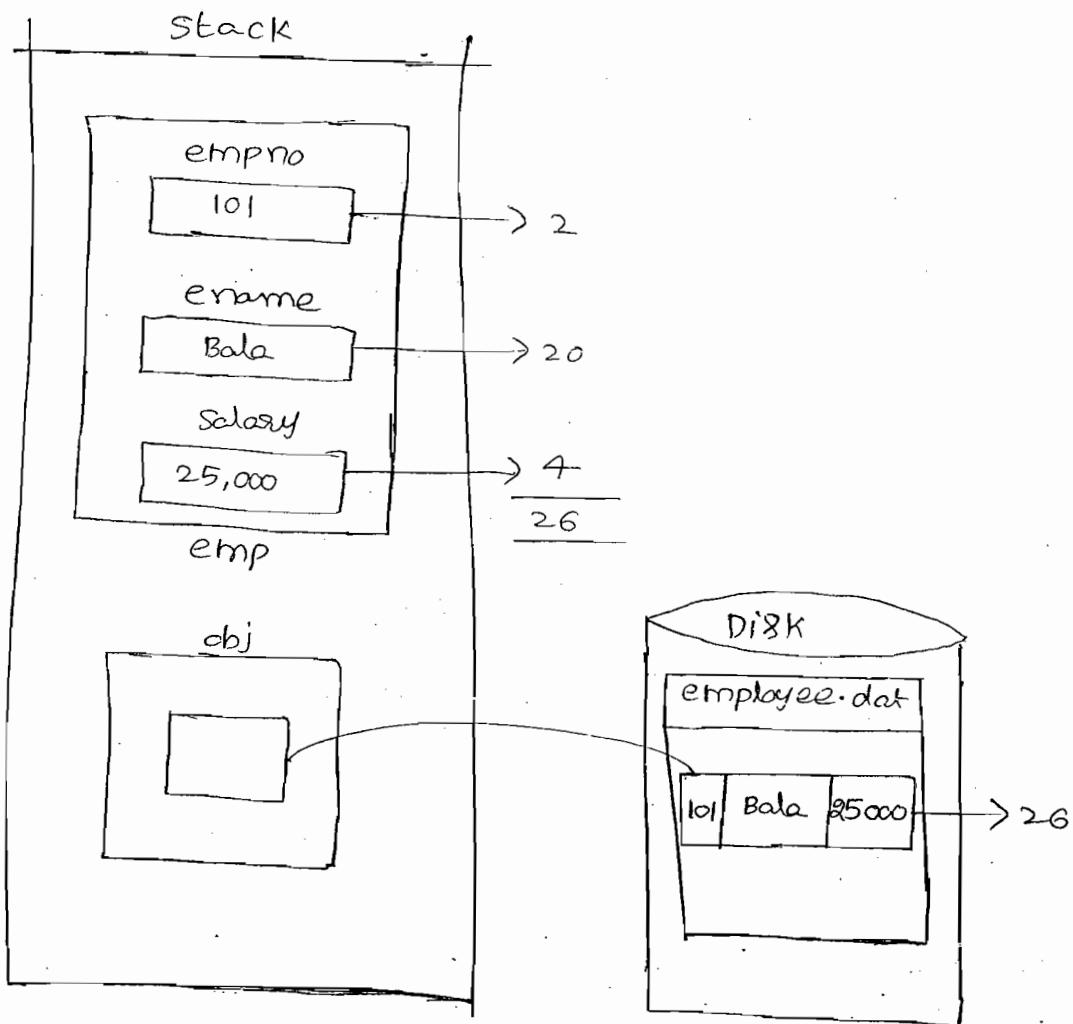
```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
class employee
{
    int empno;
    char ename[20];
    float salary;
public:
    void read()
    {
        cout << "In Input empno: ";
        cin >> empno;
        cout << "In Input employee name: ";
        cin >> ename;
        cout << "In Input salary: ";
        cin >> salary;
    }
    void print()
    {
        cout << " " << empno << "\t" << ename << "\t" << salary;
    }
};

int main()
{
    employee emp;
    ofstream obj;
    obj.open("...")
```

```

char ch='y';
clrscr();
while(ch != 'n')
{
    emp.read();
    obj.write((char*)&emp, sizeof(emp));
    cout<<"Do you want to add another employee ? ";
    cin>>ch;
}
obj.close();
return 0;
}

```



Reading objects from existing file (obj) Records from  
existing file :

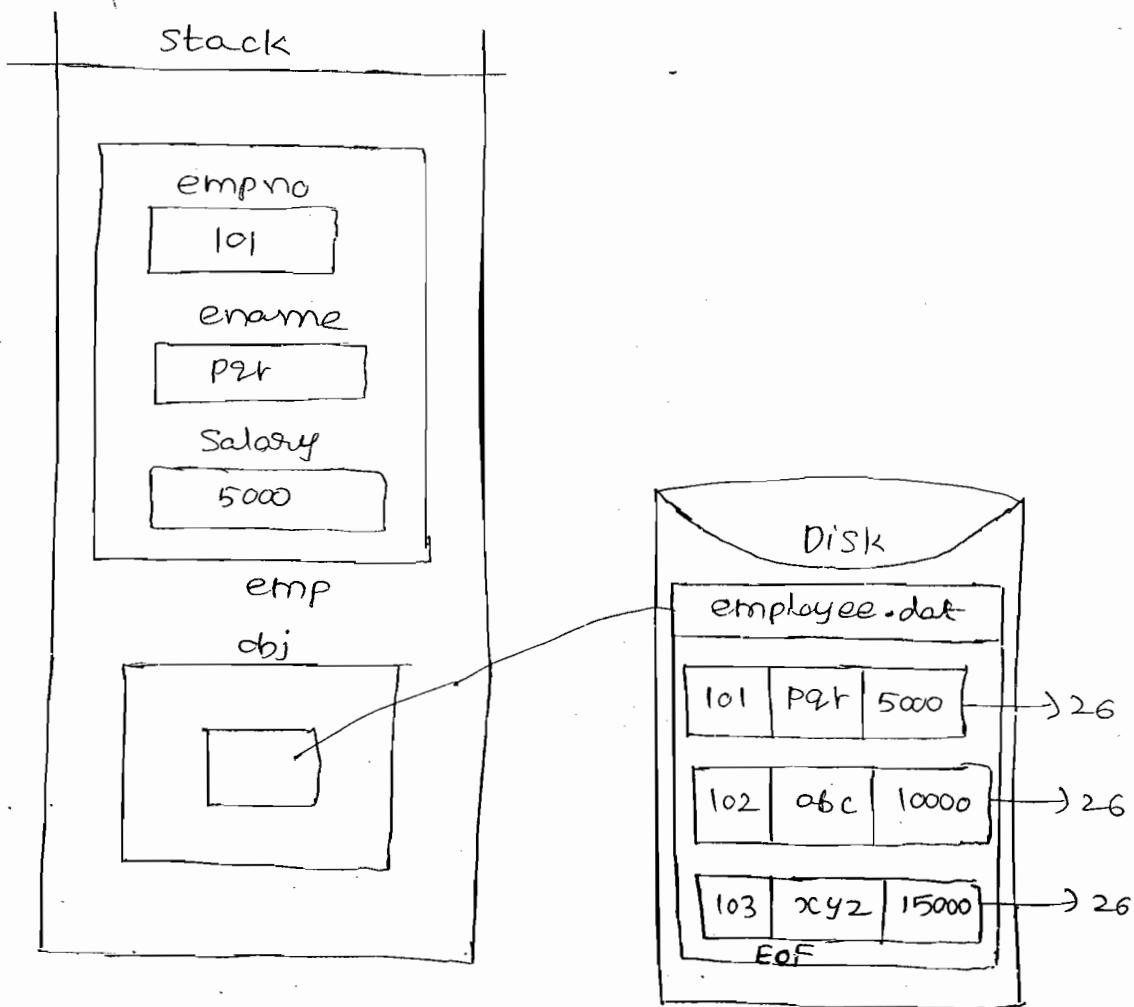
```
#include <iostream.h>
#include <iostream.h>
#include <conio.h>
class employee
{
    int empno;
    char ename[20];
    float salary;
public:
    void read();
    {
        cout << "In Input empno:" ;
        cin >> empno;
        cout << "In Input employee name:" ;
        cin >> ename;
        cout << "In Input salary:" ;
        cin >> salary;
    }
    void print();
}
cout << " " << empno << " " << ename << " " << salary;
};

int main()
{
    employee emp;
    ifstream obj;
    clrscr();
}
```

```

obj.open ("employee.dat");
while (1)
{
    obj.read (cchar*) &emp, sizeof(emp));
    if (obj.eof())
        break;
    emp.print();
}
obj.close();
return 0;
}

```



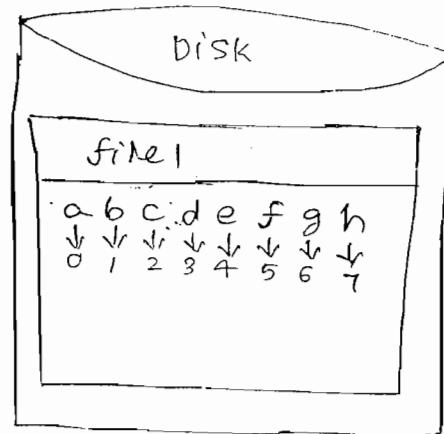
manipulation of file pointers :

- \* we can move the file pointers to any desired position in the file. we can take the control of the file pointers.
- \* The functions to position the file pointers at desired position.
- \* `seekg(Offset, position)`, `seekp(Offset, position)`,  
offset parameter represents number of bytes and  
position parameter takes one of the following three constants.

`ios::beg` `ios::cur` `ios::end`

- \* The file is open in input mode using `seekg`.
- \* The file is open in output mode using `seekp`.

```
ifstream obj;  
obj.open ("file1");  
  
obj.seekg(5, ios::beg); obj  
obj.seekg(1, ios::cur);  
obj.seekg(-2, ios::cur);  
obj.seekg(-4, ios::end);
```



tellg() :-

- \* This is member function of input stream class it will return file pointer position.
- \* It is used for to see the where file pointer located.

### fstream :

\* It is derived class of ostream and istream.

\* fstream allows to perform input and output file operations.

### open () :

\* member function which opens the file in particular mode.

### Syntax :

```
open ("filename", file-open-mode);
```

### PARAMETER :

ios:: app

### MEANING :

append to end of file

ios:: ate

goes to end of file.

ios:: binary

binary file

ios:: in

open file for reading only.

ios:: norecreate

open fails if the file does not exist.

ios::noreplace

open fails if the file already exists.

ios:: out

open file for writing.

ios:: trunc

deletes contents of the file.

\* ios is a class and out is a data member.

Ex: fstream obj;                              class data member.

```
obj.open ("Student", ios::out);
```

```
obj.open ("Student", ios::app);
```

```
obj.open ("Student", ios::out ! ios::app);
```

```
obj.open("student", ios::out | ios::nocreate),  
obj.open("student", ios::out | ios::in),  
                          ↓  
                 Reading / Writing  
                 Replace / updation.
```

Program :

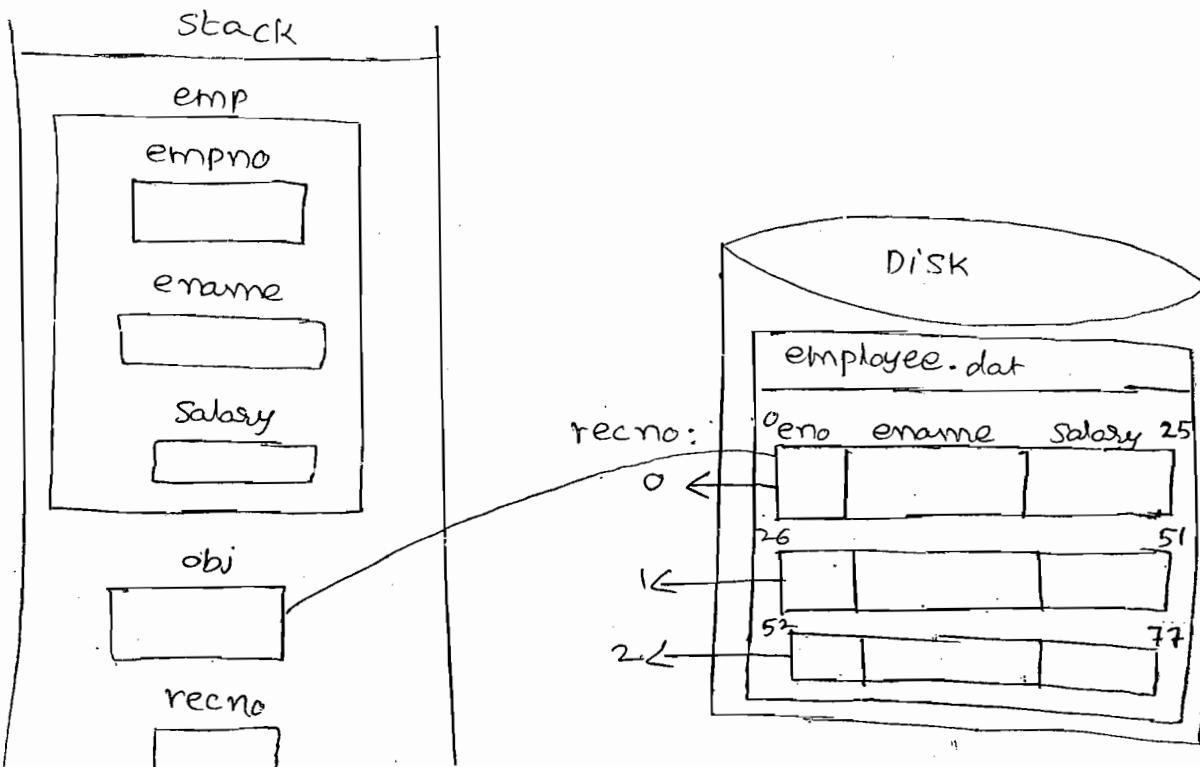
Reading employee objects randomly.

```
#include <iostream.h>  
#include <iostream.h>  
#include <conio.h>  
class employee  
{  
    int empno;  
    char ename[20];  
    float salary;  
public:  
    void read();  
{  
    cout << "In Input employee no: ",  
        cin >> empno;  
    cout << "In Input employee name: ",  
        cin >> ename;  
    cout << "In Input salary: ",  
        cin >> salary;  
}  
void print();  
{  
    cout << "\n" << empno << " " << ename << " " <<  
        salary;  
}  
};
```

```

int main()
{
    fstream obj;
    employee emp;
    obj.open("employee.dat", ios::in);
    int recno;
    cin >> recno;
    cout << "In Input record number:";
    cin >> recno;
    obj.seekg(recno * sizeof(emp), ios::beg);
    obj.read((char*)&emp, sizeof(emp));
    if (obj.eof())
        cout << "In record not found";
    else
        emp.print();
    return 0;
}

```



## Program :

### Updating/ Replacing object

```
#include <iostream.h>
#include <fstream.h>
#include <conio.h>

class employee
{
    int empno;
    char ename[20];
    float salary;
public:
    void read();
    {
        cout<<"\n Input empno: ";
        cin>> empno;
        cout<<"\n Input employee name: ";
        cin>> ename;
        cout<<"\n Input salary";
        cin>> salary;
    }
    void print();
    {
        cout<<"\n" << empno << "\t" << ename << "\t" <<
            salary;
    }
};

int main()
{
    fstream obj;
    employee emp;
    int recno;
```

```
obj.open ("employee.dat", ios::out | ios::nocreate),  
    clrsca ( ),  
cout << "Input record no to modify ",  
cin >> recno,  
emp.read ( ),  
obj.seekp (recno * sizeof (emp), ios::beg),  
obj.write ((char *) &emp, sizeof (emp)),  
obj.close ( ),  
return 0;  
}
```

ios::truncate :

- \* which open the file in truncate mode or  
delete mode
- \* If file opened in this mode operating system  
delete the complete contents of file.

## Exception Handling

### Exceptions :

- \* Exceptions are unusual conditions in a program.
- \* They may cause the program to fail or may lead to errors. They must be dealt with.
- \* There are two types of Exceptions:
  - ① Synchronous.
  - ② Asynchronous.
- \* "out-of-range index" and "over-flow" are synchronous type.
- \* Errors caused by the events beyond the control of the program like keyboard interrupt are called Asynchronous Exceptions.
- \* Exception is nothing but a simple logical error.
- \* Exception is a simple logical error given by operating system during the execution of program.

Ex:

```
int main ()  
{  
    int x,y,z;  
    cin >> x >> y;  
    z = x/y;  
    cout << z;  
}
```

Note:

Exceptions does not support Turbo 3.0

SUPPORT FOR TURBO 4.0

- \* Exception is a runtime error when it occurs operating system abnormally terminates the execution of the program.
- \* The purpose of exceptional handling mechanism is to detect and report an "exceptional circumstance".
- \* Then appropriate action can be taken. The error handling code basically consists of two segments.
- \* one to detect errors and to throw exceptions.
- \* The other to catch the exceptions and to take appropriate action.

Try, catch, throw :

\* Three keywords are used to handle exceptions.

- \* The code that has to be monitored for exceptions has to be included in try block.
- \* If any exception occurs that will be thrown by try block using the keyword throw
- \* The exception thrown by try block is caught by catch block.
- \* A try block must be followed by a catch block.

Syntax :

try

{

Business logic;

}

catch (type parameter-name) → variable-name

{ }

catch (C type parameter-name)

{

}      Error handling logic.

- \* The try block may have more than one catch statements, each corresponding to a particular type of exception.

try

{

    throw type1 arg; throw type2 arg;

}

    catch (C type1 arg)

{

        // catch block

}

    catch (C type2 arg)

{

        // catch block

}

Program :

```
#include <iostream.h>
```

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

```
int main ()
```

{

```
    int a, b, c;
```

```
    clrscr();
```

```
    cout<< "Input a, b values : ";
```

```
    cin>>a>>b;
```

try

{

    if (b==0)

```
throw o;
else
    c = a/b;
    cout << "In c = " << c;
}
catch (int x)
{
    cout << "In divide Divide Error In ";
    cout << "In " << x;
}
return o;
}
```

program :

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
int main()
{
    char uname[10], password[10],
        chscr();
    cout << "In Input Username : ";
    cin >> uname;
    cout << "In Input Password : ";
    cin >> password;
    try
    {
        if (strcmp(uname, "nit") == 0 &&
            strcmp(password, "nit") == 0)
            cout << "In welcome to my application ";
        else
```

```
    throw 'f';
}
catch (char x)
{
    cout << "In Invalid username or password";
}
return 0;
}
```

Program:

```
#include <iostream.h>
#include <conio.h>
int main ()
{
    int x;
    clrscr ();
    cout << "Input x value: ";
    cin >> x;
    try
    {
        if (x == 0)
            throw 0;
        else
            if (x < 0)
                throw 0.5f;
        else
            cout << " x = " << x;
    }
}
```

```
catch (int a)
{
    cout << "In value is zero";
}

catch (float b)
{
    cout << "In value is -ve";
}

return 0;
}
```

name Spaces :

\* name space is a collection of variables and  
datatype ( class or structures ).

\* name space is a container.

Syntax :

namespace <name space-name>;

\* namespaces does not support on Turbo 3.0 and 4.5  
Support for Dev C++

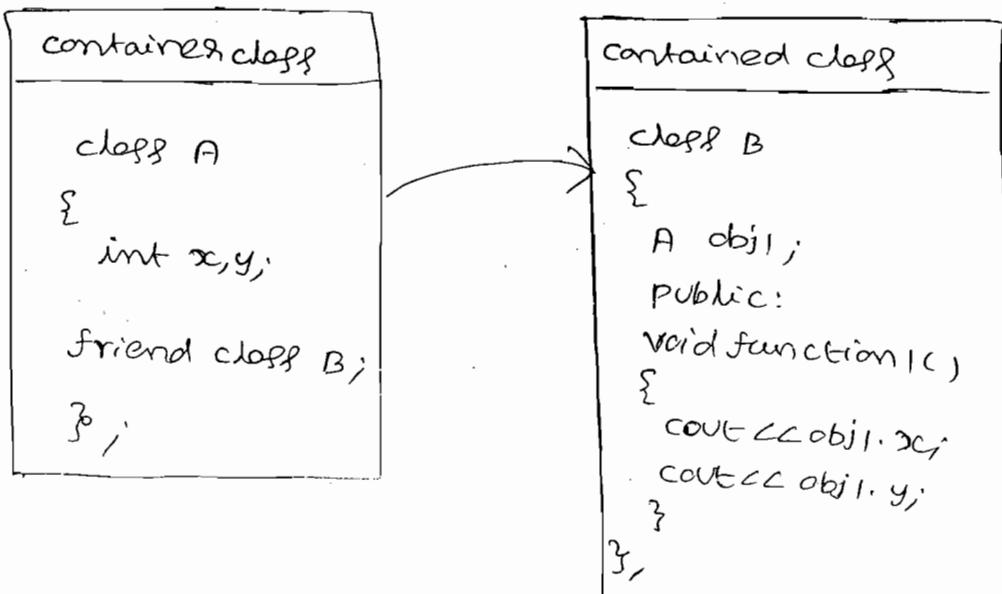
Ex:

```
namespace A
{
    int x;
}

int main()
{
    cout << A::x;
}
```

```
using A;  
int main()  
{  
    cout << x;  
}
```

Contain class and contained class:



commandline arguments:

\* The values which are send command prompt to main function is called commandline arguments.

Program:

```
#include <iostream.h>  
#include <conio.h>  
  
int main (int argc, char *arg v [ ] )  
{
```

```
class C {
public:
    void fun() {
        cout << "In Argument count " << argc;
        cout << endl << argv[1];
        cout << endl << argv[2];
        return;
    }
};
```

[To save exc]

```
C:\TCC> exc 10 20
```

OUTPUT:  
~~~~~

go to dos shell

```
C:\TCC> exc 10 20
```

Argument count 3

10  
20  
—

## Graphics

\* Graphics does not support Turbo 4.5 It supports only Turbo 3.0

initgraph :

\* init graphical screen.

initgraph(gdriver, gmode, path)

program :

```
#include <graphics.h>
```

```
int main()
```

```
{
```

```
int gd = DETECT, gm;
```

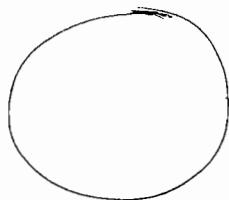
```
initgraph(&gd, &gm, "");
```

```
circle(100, 100, 50);
```

```
return 0;
```

```
}
```

output :



Program :

```
#include <graphics.h>
```

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

```
int main()
```

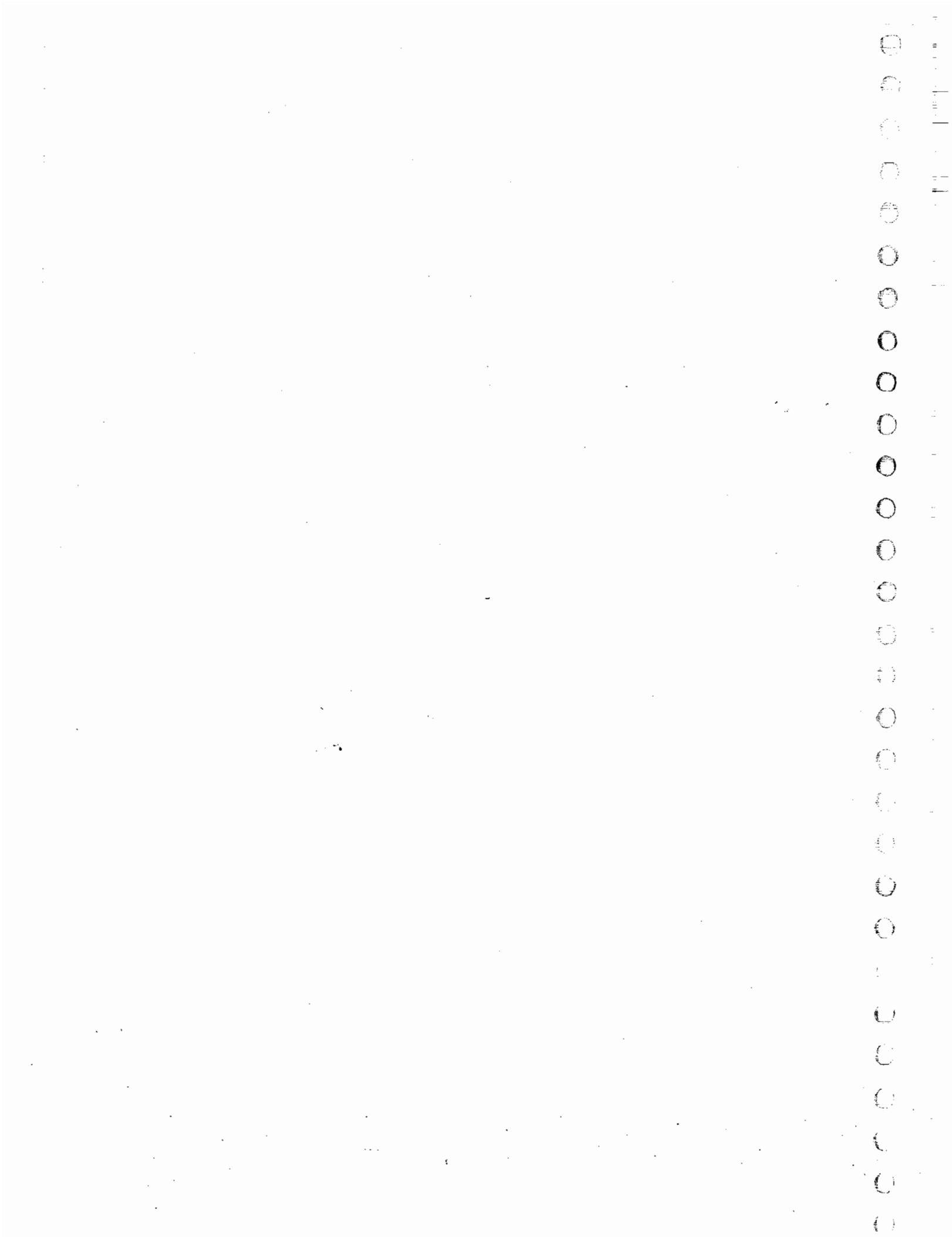
```
{
```

```
int gd = DETECT, gm;
```

```
initgraph(&gd, &gm, ""),  
for (int r=10; r<=50; r=r+5)  
{  
    setcolor(random(15)),  
    circle(getmaxx()/2, getmaxy()/2, r)  
    return 0;  
}
```

### Program:

```
#include <graphics.h>  
#include <stdlib.h>  
#include <conio.h>  
#include <dos.h>  
int main()  
{  
    int gd=DETECT, gm;  
    initgraph(&gd, &gm, "");  
    while (!kbhit())  
    {  
        delay(500),  
        setcolor(random(15)),  
        circle(random(getmaxx()), random(getmaxy()),  
              50),  
        return 0;  
    }
```



Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

Θ

GT

CO

O

+

O

S

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O



O

Θ

δ

Φ

Ω

ο

ε

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο

ο



◎ 中国古典文学名著全译本

C O C O C O C O C  
O O O O O O O O  
O O O O O O O O  
O O O O O O O O  
O O O O O O O O  
C O C O C O C O  
O O O O O O O O  
O O O O O O O O  
O O O O O O O O  
O O O O O O O O  
C O C O C O C O  
O O O O O O O O  
O O O O O O O O  
O O O O O O O O  
O O O O O O O O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

O

