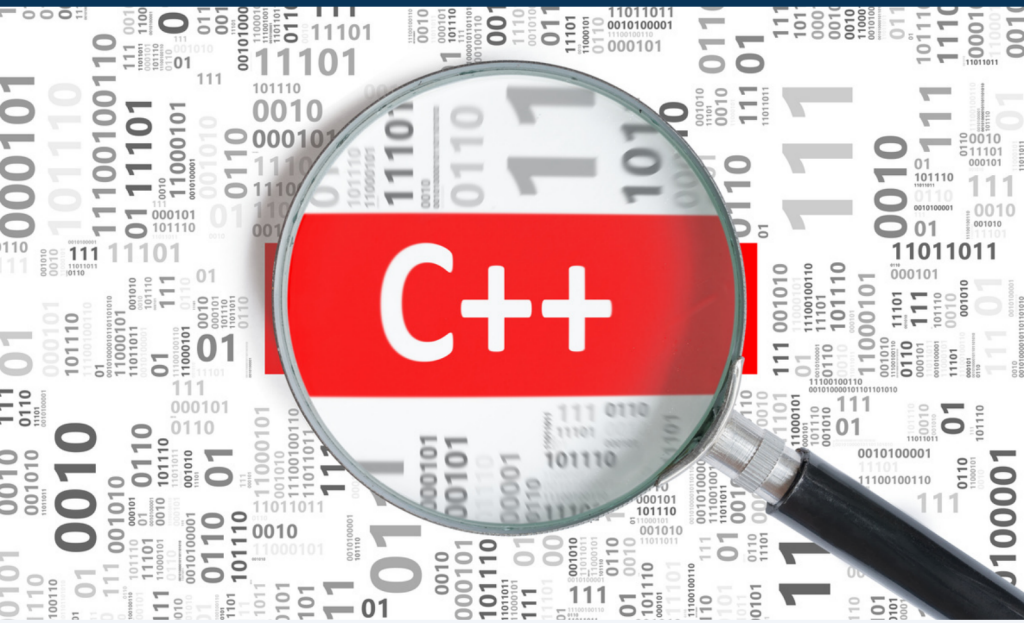


Hand written C++ Notes

CONTRIBUTOR

Santosh Kumar Mishra

SDE @Microsoft

Introduction

Q. What is C++ ?

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

Q. What is programming principle ?

A. a programming principle define set of Rules and Regulations. By following this principle complexity and developing program can be Reduce.

Programming elements :-

Programming elements are two types Those are 1. data

2. Instructions

Every program requires input, process and output.

Input :-

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

process :-

Programming some calculation is called as process.

output :-

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 principls :-

There are six types of programming principls. 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) :-

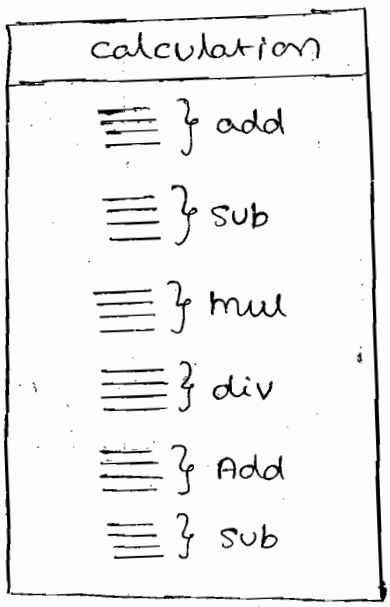
According to unstructured programming concept, program is developed by organizing data and instructions in sequential order. It is organizing sequential order.

Drawbacks of unstructured programming :-

* Redundent (Repitation) code/ instructions
* Because of Redundency the size of program increases.

* occupy more space, which reduce speed.

* There is no proper organization of data and operations.



Exm :-

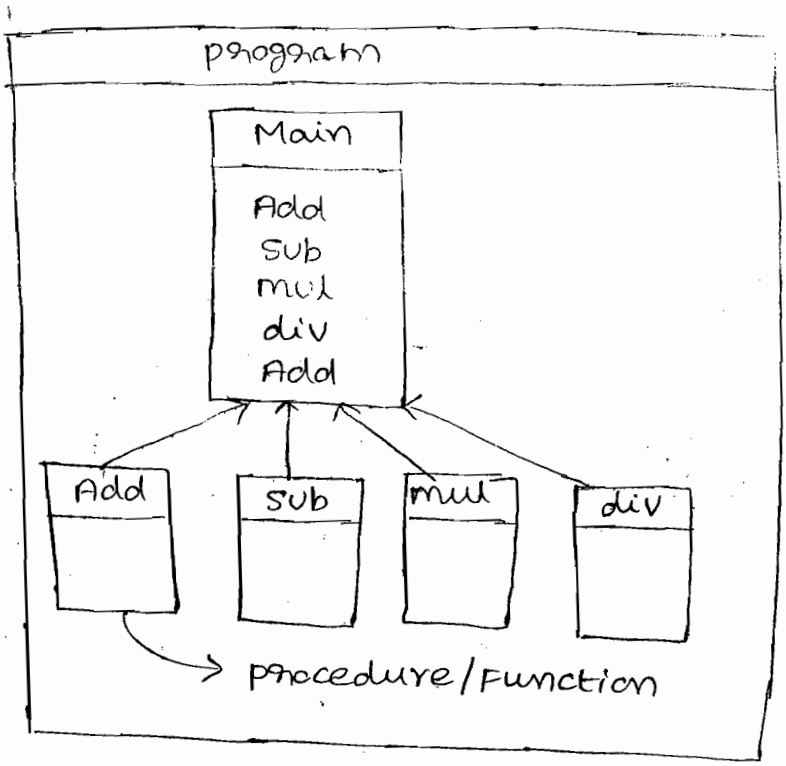
Assembly Language, machine Language are unstructured programming Languages.

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 :-

There are four types.

1. Modularity.
2. Reusability.
3. Simplicity.
4. Efficiancy.

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. Efficiancy :-

By Reducing The Size of The program Efficiancy of procedure oriented programs increases.

Drawbacks of procedural oriented programming :-

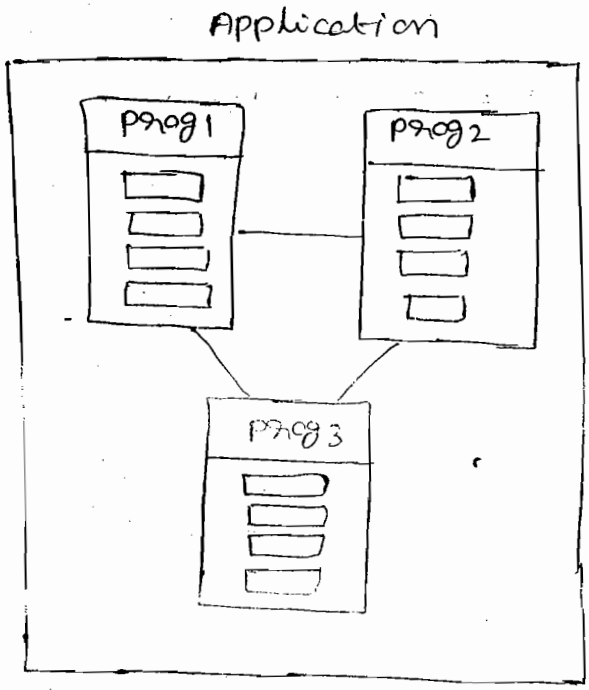
* concentrated on development of functions.
* Little attention is given to the data being used by function.
* In a Large program It is very difficult to identify what data is used by which function.
* when revising an External data, structure all The functions That Access The data also has to be Revised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

characteristics of procedural oriented programming

* Emphasis is an 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 subroutins 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 principle 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)

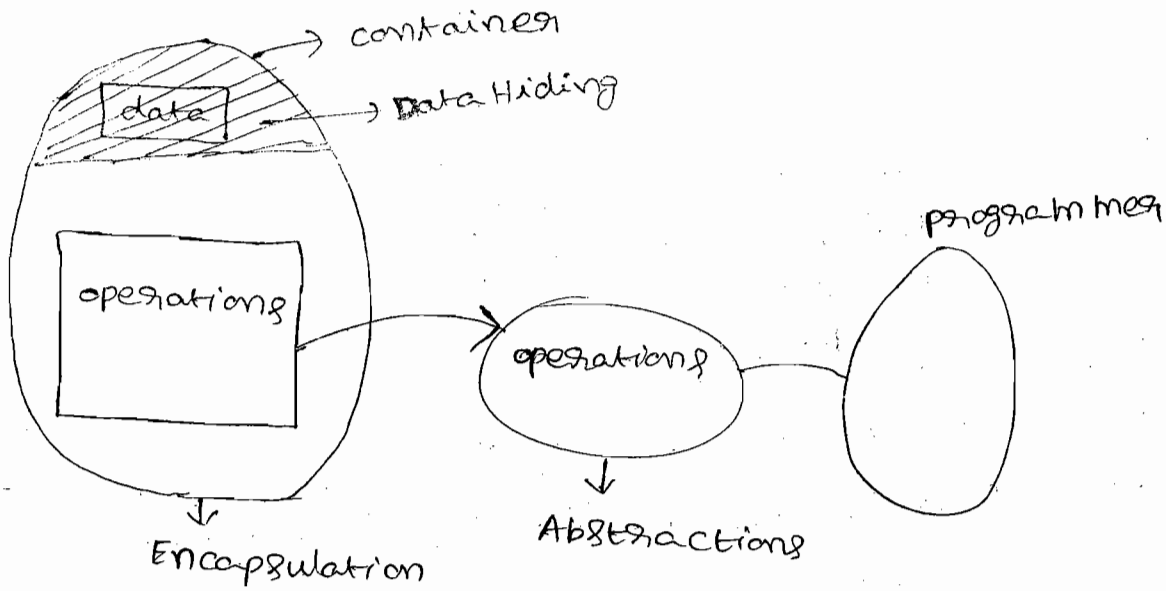
* Data is not accessible by the outside world.

* 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 programmer to build secure programs.



Abstraction :-

Abstraction is a process by which user 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 contain data and operations.

* class is Encapsulated with information and instructions.

* class is a data type/ user defined data type.

* class is a blueprint or object.

* blueprint is nothing but a planning before constructing an object.

* class is a logical view or object.

* class is planning before constructing object.

* class is a collection or members

1. data members.

2. member functions.

* class is a collection of variables and Functions.

* An user defined data type, the entire set or data and code to manipulate data.

* objects are variables or type class.

* Any number or objects or 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 :-

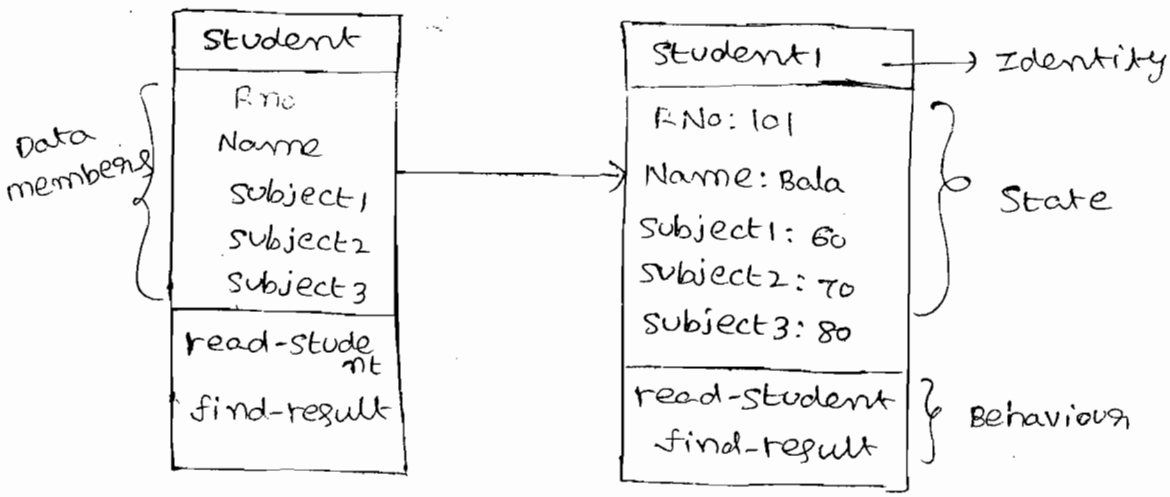
A value given to variable/attribute of object define state.

Behaviour :-

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

Identity :-

A Unig name to identify each object.



C++

* C++ is an object oriented programming Language.
* C++ is superset of C Language.
* The idea comes ++ operator.
* C++ is a 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 compatability :-

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

* C++ is an Embeded programming Language which is having The characteristics of Low level and high level Languages.

Applications of C++

Application Software :-

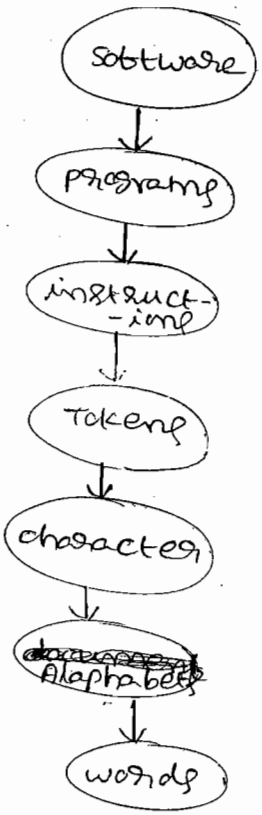
I can develop Application Softwares are

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

A computer which solve the problem of Endusers is called Application Software.

System Software :-

→ operating System.
→ compilers.
→ Devices Drivers
  - printer
  - mouse
  - keyboard.
  - monitor.



Business Application :-

→ Bank
→ Super markets
→ Hospital
→ Students school / college

communication :-

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

History of c++ :-

* 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 collections of tokens.
* A token is collection of characters.

character Set of c++ :-

* c++ uses ASCII code character set.
* According 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 casesensitive 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 There 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 :-

There 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 bytes
10. unsigned char ——— 1 byte
11. signed char ——— 1 byte
12. bool ——— 1 byte.

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

② Derived data type :-

① Array

② Pointers

③ Reference.

③ User defined datatype :-

① Struct.
② union.
③ Enum.
④ class.

④ Empty Data types :-

Void.

operators in C++ :-

The operators which are added in C++ are

1. new
2. delete
3. :: (Scope Resulution 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;

② differences between cout and printf

cout	printf
1. It is an object	1. It is an function
2. formatted and unformatted	2. formatted output
3. It doesn't specify formatted	3. It required formating specifier

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 with in I/o stream.h (iostream.h)
* This object is bind with an operator << (insertion operator), inorder to insert data with in output stream.
* insertion operator is a function available in cout object.
* cout object represents monitor.
* It is an overloaded left shift operator
* The behaviour of this operater changes when used with fundamental data types.

Syntax :

```
cout << Variable/constant;
```

Ex :-

```
cout << "Hello";
cout << 10;
```


```
cout << 10,20; ⟶ Error
cout << 10;
cout << 20;   } This is correct
```

* 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 :-

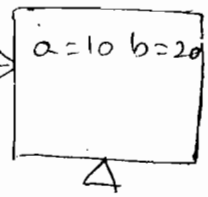
a = 10 b = 20

```
cout << a << b;
```


```
cout << "a=" << a;
```

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

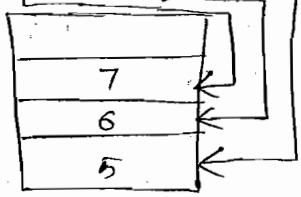
```
cout << "a=" << a << "b=" << b;
```



Ex :-

```
int a;
a = 5;
cout << a++ << a++ << a++;
```

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

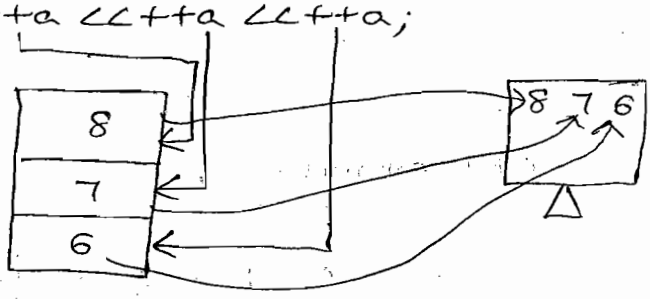


Ex :-

```
int a;
a = 5;
```

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

a = 5
++a = 6
a = 6
++a = 7
a = 7
++a = 8

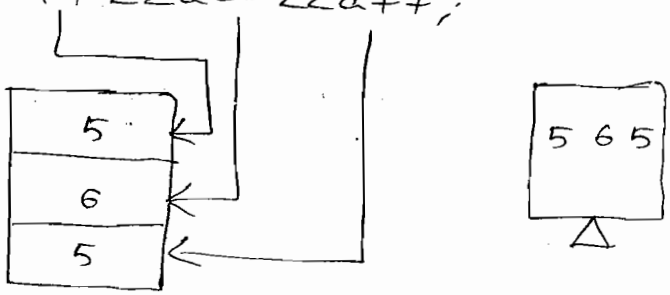


Ex :-

```
int a;
a = 5;
```

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

a = 5
a++ = 6
a = 6
a-- = 5
a = 5
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 store inside memory.

* It is an overloaded operator.

Syntax :-

```
cin >> variable;
```

* It perform formatted/unformatted input operations.

* It does not uses formatting specifiers.

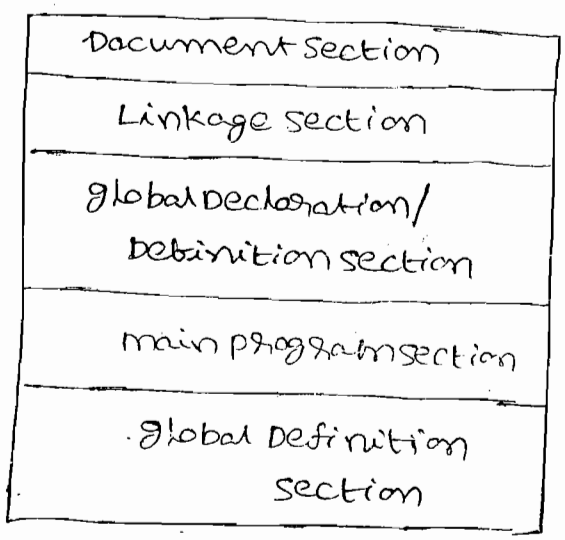
```
cin >> var1 >> var2 >> var3 ---
```

C++ compilers :-

C++ compilers available in market are

Turbo C++
Borland C++
Microsoft C++
C++ on unix
C++ on Linux
C++ on Solaries.

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 an 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. Singleline 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 The program and it is used for Linking other program with current program

* The most commonly used for preprocessor directives is

include

** (Q) What is difference between # include <file name> and # include "file name".

(A)

# include <filename>	# include "file name"
1. Search file name in include path.	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 of 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 execut the program It gives the error.

Syntax-ii :- → empty data type

```
Void main ( )
{
  ==
  ==
}
```

* The libraries does n't have the main program.
* It is an optional section.

⑤ global Definition Section :-

* Writing the functions and data type definition which have declared in global Declaredation section
* It is an optional section.

* TURBO C++ 3.0 does n't support Exception handling and name spaces 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 :-

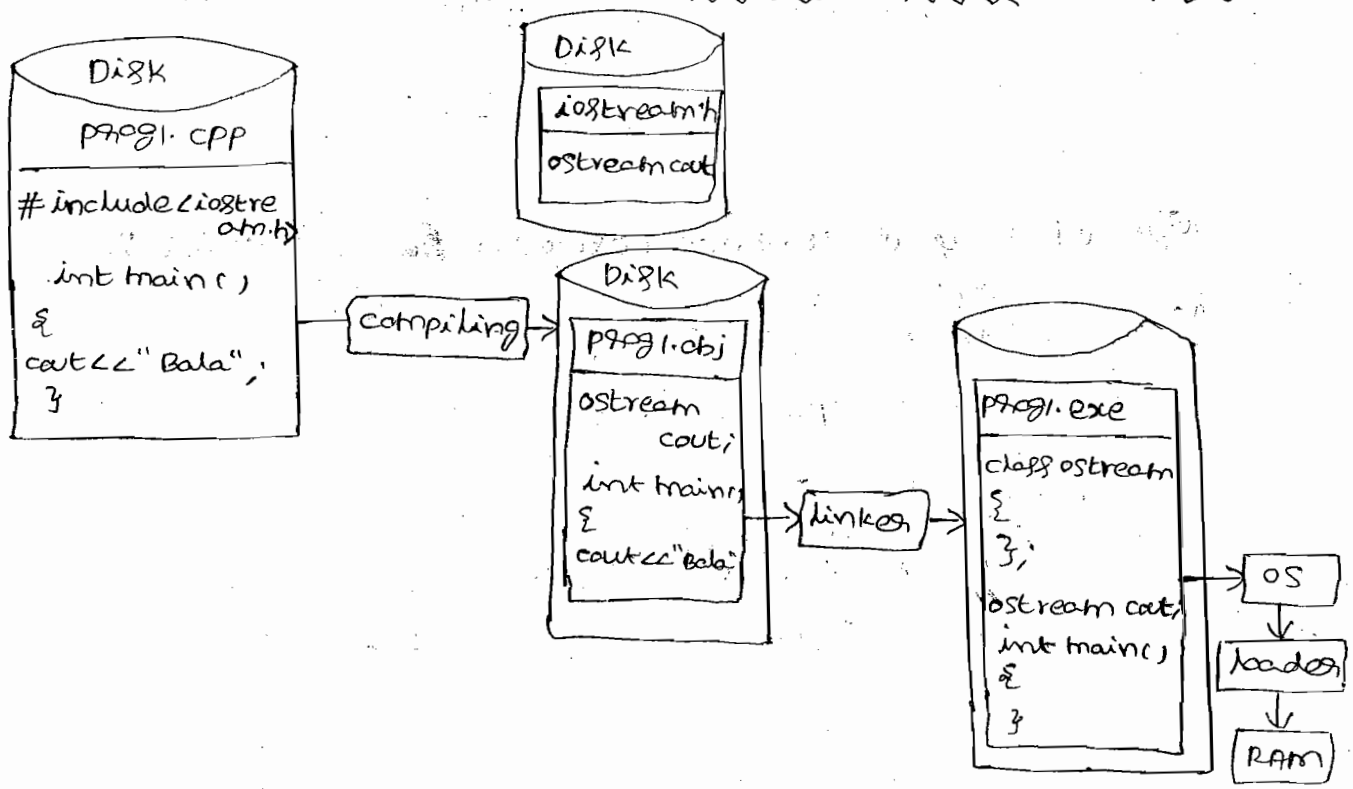
which translates object program into Executable program.

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++

The above program display compile time errors because AM 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 of space.

endl :-

→ It is a manipulator

→ It does not occupy any space.

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

Local variables :-

Q) what is variable ?

A) → variable is a container, which contains value.

→ variable is a named memory location which hold value

Q) what is Local variable ?

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

* C++ allows to declared local variables any where with in the functions.

* Program main area can be divided into 4 types.

→ Data Area.
→ Heap Area.
→ Stack Area.
→ Text Area.

Characteristics of Local Variables :-

→ Type :-
Data type which is define type of value hold by Variable.

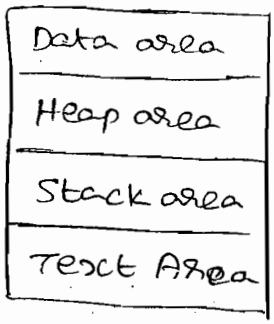
→ Scope :- The scope of local variable is with in function.

→ initial value :-
garbage/unknown

→ life time :-
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 variables or, auto variable
* auto is a storage 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 :-

* 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 as Static initialization

Syntax :-

datatype variablename = constantValue;

Ex :- int x = 10;

* A value assigned to the 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

O/P
Base 1.5
Height 2.5
Area 1.875

```
#include <iostream.h>
#include <conio.h>
  int main()
  {
   float base=1.5, height=2.5; → static initialization
     cout<<"\n Base "<<base;
     cout<<"\n Height"<<height;
     float area = 0.5*base*height; → Dynamic initialization
  cout<<"\n area of triangle"<<area;
     return 0;
   }
```

program :-

// write a program to read name, three subjects marks and calculate total and Average.

```
# include <iostream.h>
# include <conio.h>
  int main()
  {
    char name[10];
    int sub1, sub2, sub3;
      clrscr();
```

```
cout << "\n Input name";
cin >> name;
cout << "\n Input 3 subject masks";
cin >> sub1 >> sub2 >> sub3;
int total = sub1 + sub2 + sub3;
float avg = total/3.0;
cout << "\n Name" << name;
cout << "\n Total" << total;
cout << "\n 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 formated 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. Annamous block.

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

* The variables declared inside the rest 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.

life time :- until the block is executed.

initial value :- garbage

Storage :- Stack.

* 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 seccured

* c++ is not a pure object oriented because it support 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

life time :-

until program is executed

Storage :-

Data type

initial value :-

int — 0

float — 0.0

char — null (non printable characters).

↓

\0

* 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 revelation operator (::) :-

If local and global variables with same name global variable is refered with in function using scope revelation operator (::)

Q. 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

Q. 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 numerics they will be displayed right justified.

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

output : 123

Ex:-
```
# include <iostream.h>
# include <conio.h>
# include <iomanip.h>
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;
}
```

o/p
```
  10
 200
3000
```

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 characters

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

(a) Find output ?

```
#include <iomanip.h>
#include <conio.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 << setios flags(ios::right);
cout << setios flags(ios::left);

* used for displaying float field in fixed or scientific notation

Ex :- cout << setiosflags(ios::fixed);
cout << setios flags(ios::scientific);

* used for displaying trailing zeros in a float field.

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

* used for displaying positive sign.

Ex: cout << setiosflags(ios::showpos);
↓ ↓
class Data manipulator

Program:

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

o/p

10.000000

* Short-point displays the precision (default number of precision)

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 integers can't used with floats.

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 in one by one C 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.

* These 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.

* These 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)

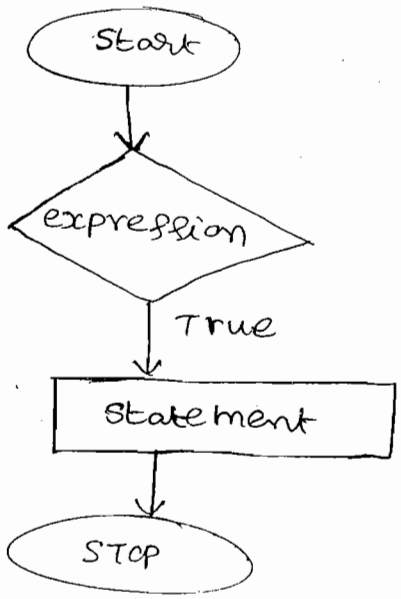
Syntax :

(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 <[redacted]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<<" Eligble for vote"<<"\t"<<name;
  return 0;
}
```

output :

```
Input Name : bala
 Input Age : 22
Eligble for vote bala.
```

* if statement can be terminated with semicolon (;) it does not have else block.

* if is terminated with semicolon means it does not have any statement is 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 if 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
    statement2;
```

* if expression return non-zero (true) execute statement 1.

* If expression return zero (false) execute statement 2.

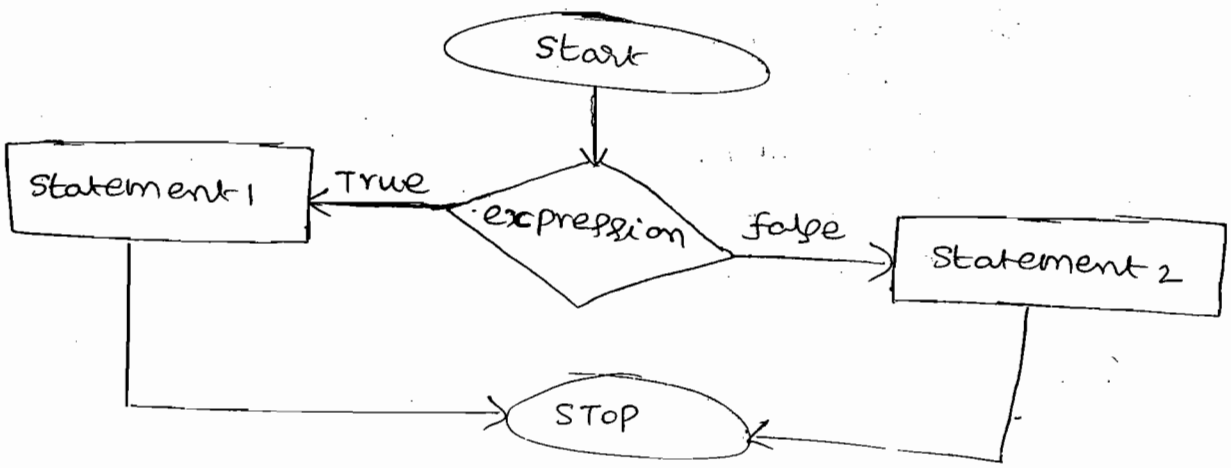
Syntax :

```
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<< "\n not Vowel ";
    return 0;
}
```

output :

```
Input Any character : A
  Vowel
Input Any character : a
  not Vowel.
```

```
if (expression)
{
   statement1;
   statement2;
   statement3;
}
```

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

* If expression is false if statement1 is skip and executes statement2 and statement3.

program :

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

```
#include<iostream.h>
#include<conio.h>
int main()
{
    int num;
    clrscr();
    cout<<"\n 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<< "\n 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

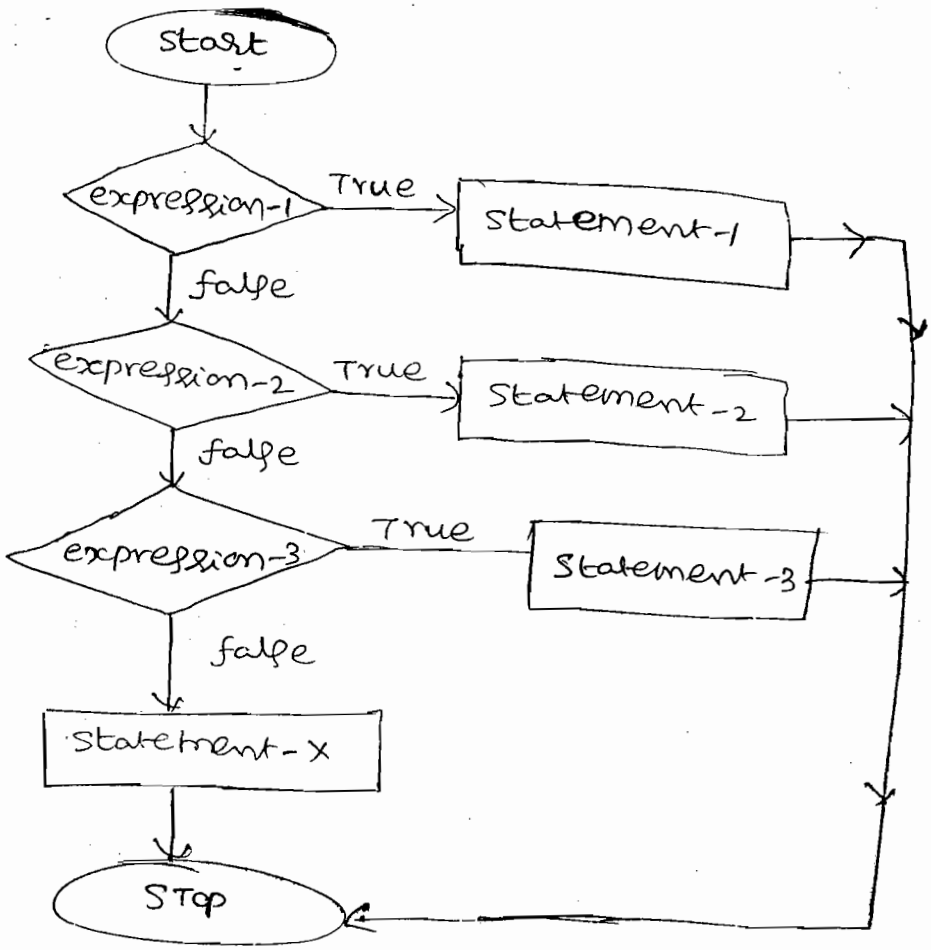
if-else-if ;

* 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<<"\n Input a,b,c:";
cin>>a>>b>>c;
if(a>b && a>c)
cout<<"\n a is maximum:"<<a;
else
 if(b>c)
  cout<<"\n b is maximum:"<<b;
  else
  cout<<"\n c is maximum:"<<c;
   return 0;
}
```

output : Input a, b, c values : 10 20 30

c is maximum : 30

Input a, b, c values : 10 5 2

a is is maximum : 10

program :

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

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

output :

Input Any character : A

alphabet

Input Any characters : 5

digit

Input Any character : &

Nested if :-

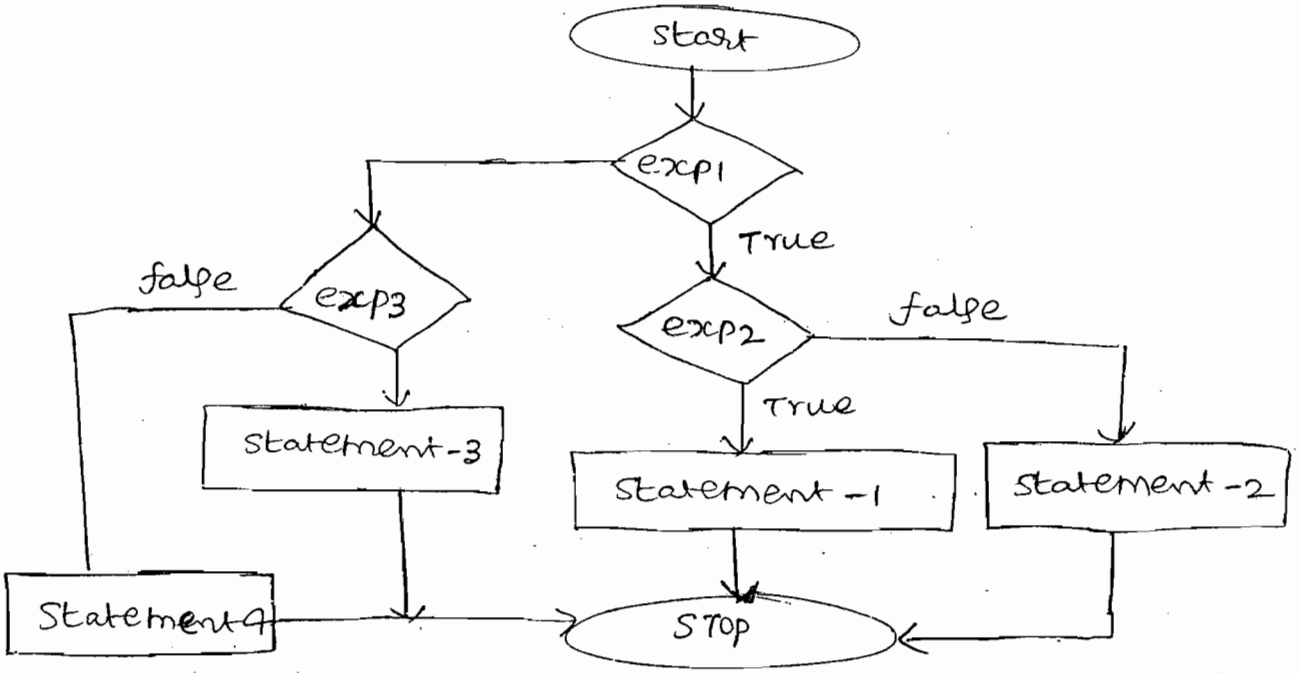
* Nested means with in or immediately.

* 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<<"\n Input Name:";
    cin>> name;
    cout<<"\n Input 3 Subjects:";
    cin>> sub1 >> sub2 >> sub3;
    if (sub1>=40)
    if (sub2>=40)
    if (sub3>=40)
    cout<<"\n pass";
     else
     cout<<"\n fail";
     else
      cout<<"\n fail";
      else
       cout<<"\n fail";
        return 0;
}
```

output :

```
Input Name : Bala
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:
        statement1;
       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 to 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 any where with 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 any where

Program :

Write a program to display the following menu

1. Area of the Rectangle,
2. Area of the Square
3. Area of the Traingle
4. Area of the parallogram
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<<"\n 1. Area of the Rectangle";
cout<<"\n 2. Area of the Square";
cout<<"\n 3. Area of the Traingle";
cout<<"\n 4. Area of the parallogram";
cout<<"\n 5. Area of the Rombus";
cout<<"\n 6. Area of the circle";
cout<<"\n Input your option:";
cin>> opt;
  switch(opt)
  {
```

```
case 1:
        cout<<"\n Input length and breadth:";
        cin>> length >> breadth;
        area = length * breadth;
        cout<<"\n Area of The Rectangle:" << area;
        break;
case 2:
        cout<<"\n Input side:";
        cin>> side;
        area = side * side;
        cout<<"\n Area of The Square:" << area;
        break;
case 3:
        cout<<"\n Input base and height:";
        cin>> base >> height;
        area = 0.5 * base * height;
        cout<<"\n Area of The Traingle:" << area;
        break;
case 4:
        cout<<"\n Input base and height:";
        cin>> base >> height;
        area = base * height;
        cout<<"\n Area of The parallogram:" << area;
        break;
case 5:
        cout<<"\n Input d1 and d2:";
        cin>> d1 >> d2;
        area = 0.5 * d1 * d2;
        cout<<"\n Area of The
```

```
case 6:
        cout<<"\n Input r:",
        cin>> r,
        area = 3.14 *r *r;
        cout<<"\n Area of the circle: "<<area,
        break,
        default:
        cout<<"\n Invalid option",
    }
    return 0,
}
```

output :

Input your option : 6

Input r : 7

Area of the circle : 153.860001

Input your option : 4

Input base and height : 15 12

Area of the parallogram : 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;
  clrscr();
 cout<<"\n Input Any character:";
 cin>>ch;
 switch(ch)
{
 case 'A':
 case 'E':
 case 'I':
 case 'O':
 case 'U':
        cout<<"\n vowel";
        break;
        default:
        cout<<"\n not vowel";
        }
       return 0;
}
```

output

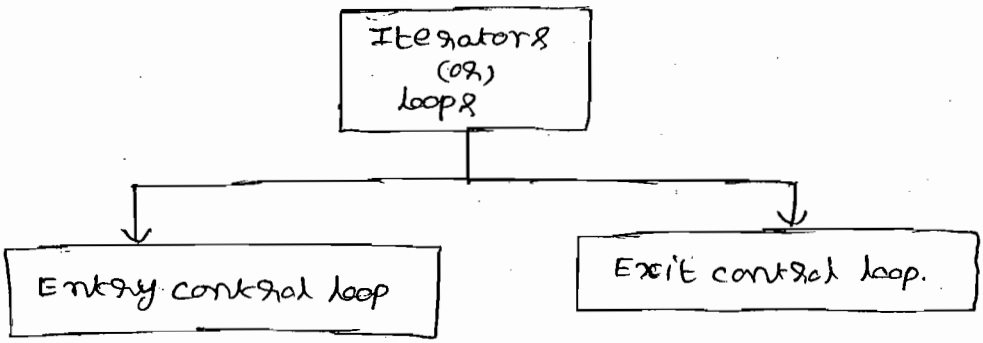
```
Input Any character : O
 Vowel
Input Any character : a
  Not vowel
```

Loops or 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 or iterators.

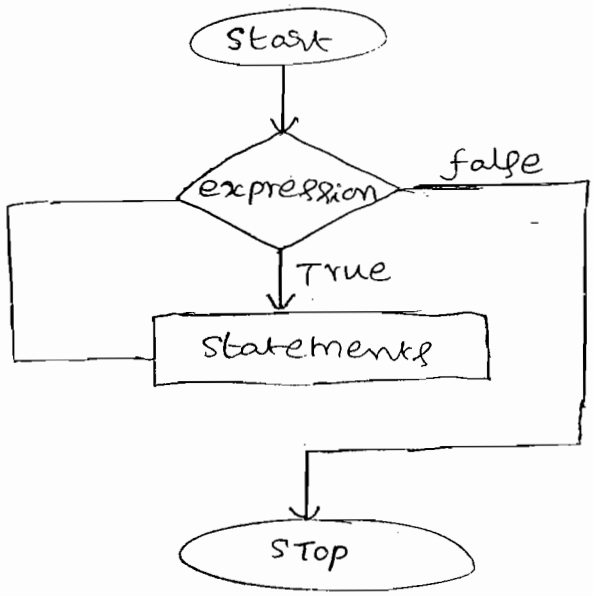
1. Entry control loop.
2. Exit control loop.



1. Entry control loop :

In Entry control loop condition or Expression is Evaluated before Executing statements.

flow chart :



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

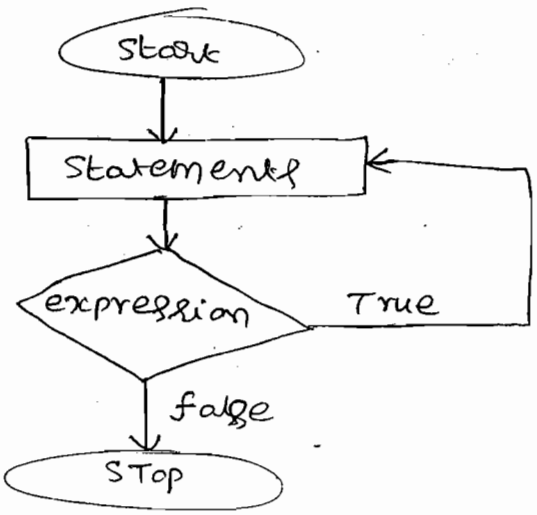
① while loop.

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 define 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 :

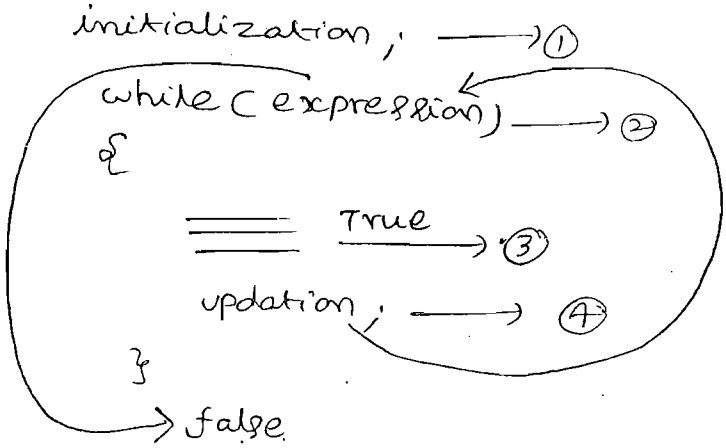
```
while (expression)
 simple - statement ;
```

(or)

Syntax :

```
while (expression)
{
   compound statements;
}
```

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

output

Sum is : 5050

program :

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

$1^2+2^2+3^2+4^2+5^2+---+20^2$

```
# 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 << "\n sum is : " << sum;
    return 0;
}
```

output

sum is : 2870

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 ( );
```

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

output :

sum is : 3025

program :

write a program sum of square 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;
    clrscr ();
    while (num <=20)
    {
       if (num %2==0)
        sum = sum+(num*num);
          num = num+1;
     }
     cout<< "\n sum is: " << sum;
      return 0;
   }
```

output

sum is : 1540

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 ( num ! =0)
 {
     r = sum %10;
    sum = sum + (r*r*r);
     num = num/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 palindrum 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 palimdrum";
 else
 cout<<"\n Not palimdrum";
  return 0;
}
```

output : Input Any Number : 121

palimdrum

Input Any Number : 125

Not palimdrum.

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<<"\n Input Any Number:";
   cin>> num;
   while(i<=num)
   {
      if( num %i ==0)
        count++;
         i=i+1;
    }
   if (count ==2)
   cout<<"\n prime Number";
    else
     cout<<"\n 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;
```

```
    clrscr();
    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 :

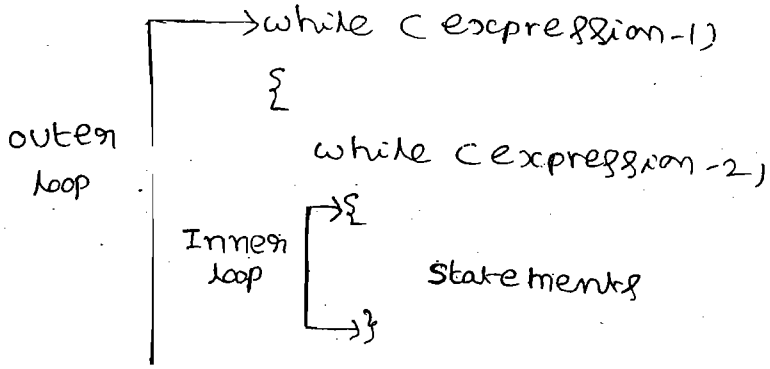
Input Any Number : 5

Factorial is : 120

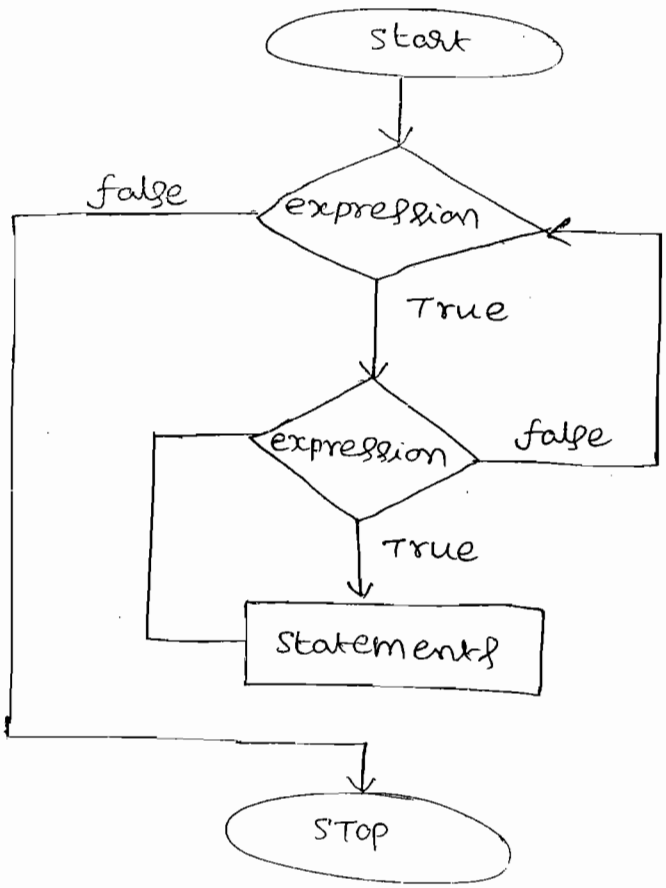
Nested While :-

* 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)
        {
```

output

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

```
        count ++;                                   53
         i++;                                       59
      }                                             61
      if ( count == 2)                              67
                                                    71
     cout<<"\n"<< num;                              73
      num = num +1;                                 79
   }                                                83
    return 0;                                       89
}                                                   97
```

program :

write a program to display the tables up 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.

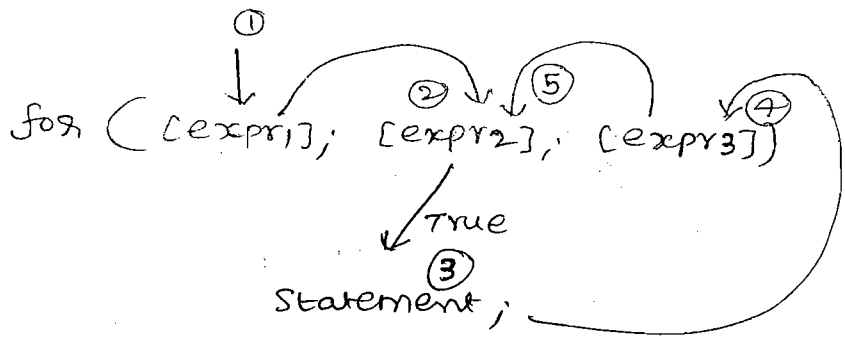
expression-2 :

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 updation 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<<"\t"<<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)
      [redacted]
     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 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 :

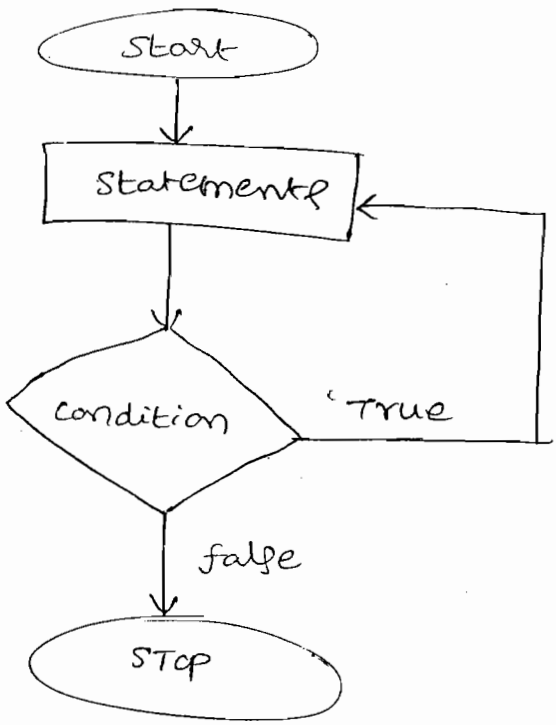
* do-while is called exit control loop, where statements are executed first and Evaluate the condition.

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<<"\n 1. Area of The Rectangle";
 cout<<"\n 2. perimeter of the Rectangle";
 cout<<"\n 3. Area of The Square";
 cout<<"\n 4 perimeter of the Square";
 cout<<"\n 5. Area of the Traingle";
 cout<<"\n 6. Area of the circle";
 cout<<"\n 7. circumference of the circle";
cout<<"\n 8. Area of the parallogram";
cout<<"\n 9. Area of the sphere";
cout<<"\n 10. volume of the sphere";
cout<<"\n Input your option:";
cin>> opt;
 switch(opt)
{
  case 1:
        cout<<"\n Input l and b values:";
        cin>> l>> b;
        Area = l*b;
      cout<<"\n Area of the Rectangle:"<<area;
        break;
```

```
case 2:
        cout<<"\n Input l and b values:";
        cin>>l>>b;
        perimeter = 2*(l+b);
        cout<<"\n perimeter of The Rectangle:"
                                        <<perimeter;
        break;
case 3:
        cout<<"\n Input  s value:",
        cin>> s;
        area = s*s;
      cout<<"\n Area of The Square:"<<area;
        break;
case 4:
      cout<<"\n Input s value:",
        cin>>s;
        perimeter =4*s;
    cout<<"\n perimeter of the Square:"<<perimeter;
      break;
case 5:
      cout<<"\n Input b and h values:";
       cin>>b>>h;
        area = 0.5*b*h;
     cout<<"\n Area of The Traingle:"<< area;
        break;
case 6:
        cout<<"\n Input r value:";
         cin>>r;
          area = 3.14*r*r;
       cout<<"\n Area of The circle:"<<area;
         break;
```

```
        case 7:
                cout<<"\n Input r value:";
                cin>>r;
                circumference = 2*3.14*r;
            cout<<"\n circumference of the circle:"<< circumfer-
                                                    -ence;
                break;
        case 8:
            cout<<"\n Input b and h value:";
            cin>>b>>h;
             area = b*h;
          cout<<"\n Area of the parallogram:"<< area;
            break;
        case 9:
            cout<<"\n Input r value:";
            cin>>r;
             area = 4*3.14*r*r;
           cout<<"\n Area of the sphere:"<< area;
             break;
        case 10:
                cout<<"\n Input r value:";
                cin>>r;
                volume = 4/3*3.14*r*r*r;
          cout<<"\n volume of the sphere:"<< volume;
             break;
    }
  } while(opt<=10);
    return 0;
}
```

break : -

* break is a keyword.
* It passes the control.

Syntax : break;

* The break statement causes control to pass to the statement following the inner most 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 transfered 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(40,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;
return 0;
}
```

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

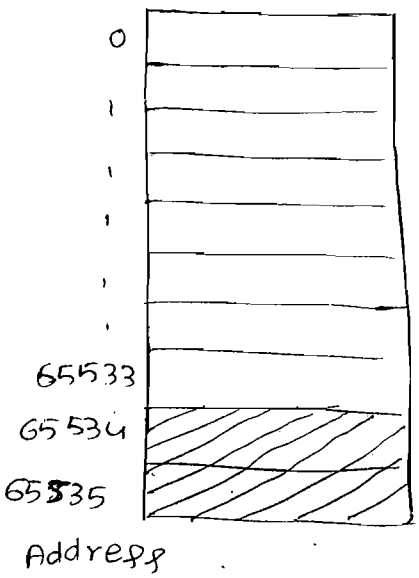
Pointers :-

* 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.

@ 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 using ampersent (&) operation

It means address of operator

cout << &a; ⟶ 65534

cout << &b; ⟶ 65532.

→ ampersent 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

→ "*" indicates pointer type.

→ '*' indicates value at address.

→ '*' indicates dereferencing operator.

→ '*' indicates point to address.

Ex:- int *a;

a=10; ⟶ Error

Advantages :-

1. Dynamic memory allocation (DMA).
2. It increases efficiancy of the program (avoiding wastage of memory)

3. Helpful building complex data types like Arrays 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 seccure.

* The size of the pointer various from one compiler to another. 16-bit compiler allocates 2-bytes for pointer. 32-bit compiler allocates 4-bytes for pointer.

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

program :

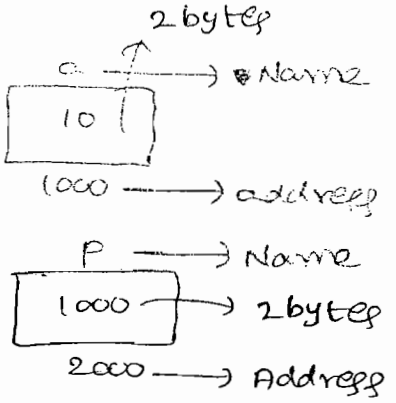
```
#include <iostream.h>
#include <conio.h>
int main()
{
    int *p;
    char *q;
    float *r;
    clrscr();
    cout<<"\n size of integer pointer"<<sizeof(p);
cout<<"\n Size of character pointer"<<sizeof(q);
cout<<"\n Size of float pointer"<<sizeof(r);
    return 0;
}
```

o/p

2-bytes
2-bytes
2-bytes

→ * is 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 define the type of value refered by pointer.

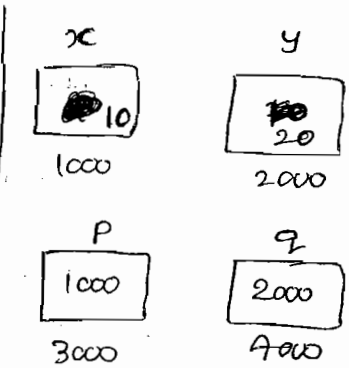
(2) Find output ?

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

The above program display compile time error because the float pointer can not hold address of integer value

Program:-

```
#include <iostream.h>
#include <conio.h>
int main()
{
   int x, y;
   int *p, *q;
   clrscr();
   cout<<"\n input x, y values";
   cin>>x>y;
     p = &x;
     q = &y;
    *p = *p + *q;
    *q = *p - *q;
    *p = *p - *q;
  cout<<"\n After Swaping \n";
   cout<<"\n x = "<<x;
   cout<<"\n y = "<<y;
    return 0;
  }
```



*p = *p + *q

Value at 1000 = Value 1000 + Value at 2000

= 10 + 20 = 30

*q = *p - *q

Value at 2000 = Value at 1000 - Value at 2000

= 30 - 20 = 10

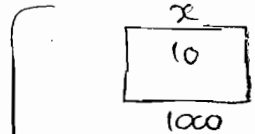
*p = *p - *q

Value at 1000 = Value at 1000 - Value at 2000

= 30 - 10 = 20

Program:-

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



*(&x)

*(1000)

↓

Value at 1000 = 10

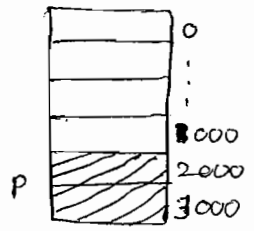
Q what is Dangling pointer ?

A → A pointer variable which hold an address of unreserved memory location is called dangling pointer.

→ These pointers are not secured.

→ This leads to memory leakage.

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



→ This dangling pointers are eliminated to programming Language like Java and .Net

Void pointer :-

* 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 can not 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 of 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;
    x=20;
    return 0;
}
```

* The above program displays compile time error because constant pointer cannot change the value of x.

Double pointers :-

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

Syntax :-

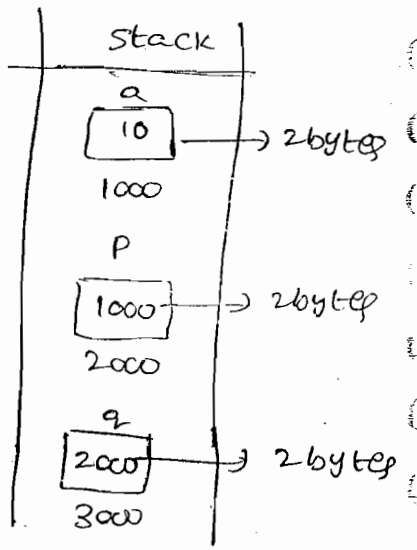
```
datatype **variable-name;
```

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;
}
```

o/p

a=10
a=10
a=10



*p
value at 1000 = 10

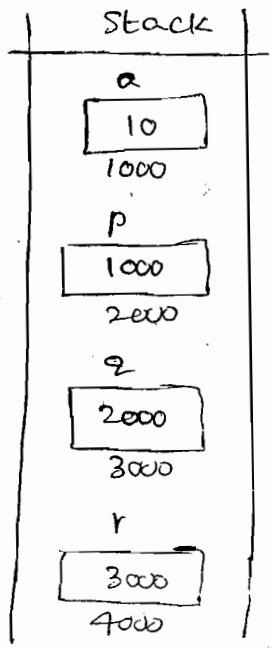
**q
Value at (Value at 2000)
Value at (1000) = 10

Program:

```
#include<iostream.h>
#include<conio.h>
int main()
{
   int a=10;
   int *p, **q, ***r;
   clrscr();
    p=&a;
    q=&p;
    r=&q;
  cout<<"\n a="<<a;
  cout<<"\n a="<<*p;
  cout<<"\n a="<<**q;
  cout<<"\n a="<<***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 refered 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 dimentional Array.
2. multi dimentional Array.

Single dimentional Array:-

* An array whose elements are refered using one subscript.
* Single dimentional 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.

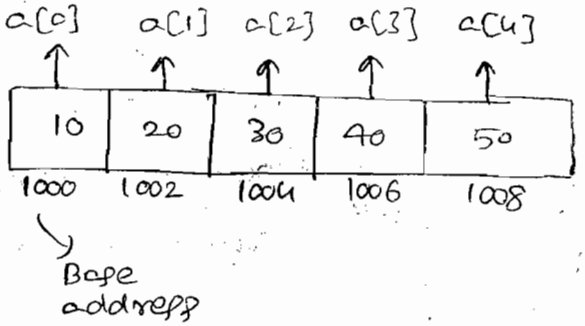
Eg:-

int a[5];

↓

array name

pointer name



*(baseaddress + offset * sizeof datatype)

a[0] = 10 → *(a + 0*2) → *(1000 + 0) → *1000

a[1] = 20 → *(a + 1*2) → *(1000 + 2) → *1002

* The formula to refer each location in array is

*(baseaddress + offset * sizeof datatype)

Eg:-

a[2] = 30

→ *(a + 2*2) → *(a + 4) → *(1000 + 4) → *(1004) = 30

Comments:-

→ Each location in array identify a unique number is called index number.

→ Index number is logical number.

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;
    clrscr();
  cout << "\n input name";
   cin >> name;
  cout << "\input 3 subjects";
  for (index=0; index<3; index++)
   cin >> subject[index]; // * (subject+index)
   cout<<"\n name " << name;
   for (index=0; index<3; index++)
    cout<<"\n" << 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;
   clrscr();
  cout << "\n input how many elements?";
   cin >> n;
    cout<<"\n Input elements";
    for(index=0; index<n; index++)
      cin>>a[index];
```

```
cout << "\n 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[9]; → 10, 20, 30, 40, 50, 60, 90

⇒ int a[5] → value

int a[5] — single pointer

int *a[5] — double pointer

Array of pointers * :-

* It is an array which hold 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
1000	2000	3000

[*a[0] = *(*(a+0)) = *(*

Double dimensional array :-

* 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]; (5 → rows, 3 → column)

int runs[11]; ⟶ int runs[5][11];

* It is a implicit double pointer.

Syntax :-

datatype array name[rowSize][colSize];

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 << "\n Input elements of matrix";
   for (i=0; i<3; i++)
   for (j=0; j<3; j++)
   cin >> a[i][j]
   cout << "\n Elements of matrix\n";
   for (i=0; i<3; i++)
   {
     for (j=0; j<3; j++)
      cout << *(*(a+i)+j);
      cout << "\n";
      }
      return 0;
   }
```

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 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<<"\n input how many subjects ? ";
  cin >> n;
  cout<<"\n Input marks \n";
    for (i=0; i<n; i++)
     for (j=0; j<3; j++)
      cin >> student[i][j];
      cout<<"\n marks are \n";
    for (i=0; i<n; i++)
     {
       for (j=0; j<3; j++)
        cout<< student[i][j]<<"\t";
          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<<"\n Input a matrix:" <<"\n";
 for(i=0; i<3; i++)
 for(j=0; j<3; j++)
  cin >> a[i][j];
 cout<<"\n Input b matrix:" <<"\n";
 for(i=0; i<3; i++)
 for(j=0; j<3; j++)
 cin>> b[i][j];
 cout<<"\n Adding of two matrices:"<<"\n";
 for(i=0; i<3; i++)
{
 for(j=0; j<3; j++)
 cout<<" " << a[i][j]+b[i][j];
  cout<<"\n";
  }
  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] = C[i][j] + A[i][k] * B[k][j];
        }
        cout << "  " << C[i][j];
    }
        cout << "\n";
    }
    return 0;
}
```

output :

Input Elements of A matrix:

1 2 3
4 5 6
7 8 9

Input Elements of B matrix :

1 0 0
0 1 0
0 0 1

matrix multiplication:

1 2 3
4 5 6
7 8 9

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

Dynamic memory allocation (DMA) :-

* "Dynamic" term refers of runtime.
* allocating or reserving 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 :-

<pointer_to_name> = new <name>[<name_initializer>];

The new operator tries to create an object or Variable name by allocating size of (<name>) bytes in the heap.

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

	int ↓	float ↓	'char ↓
intial values	0	0.0	non printable value
Storage	Heap area		
life time	until delete, until execution of the program		

② what is difference between new and Malloc ?

new	malloc
1. It is a operator.	1. It is a function
2. It allocates memory for one or more than one value	2. It allocates memory for only one value
3. It does not required any type casting	3. Return type do this function is void pointer, It required type casting.

2. delete :-

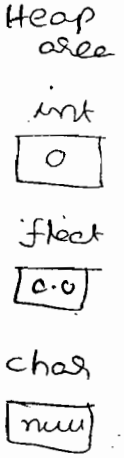
Syntax :-

delete <pointer-to-name>;

The "delete" operator destroys the object or variable <name> by deallocating sizeof(<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;
}
```

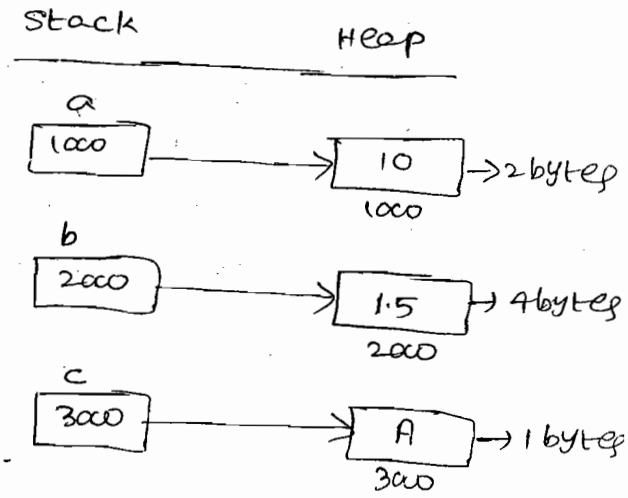


* The new operator reserve memory, and return address of that memory location.

* Address of dynamically allocated variable hold by a pointer variable.

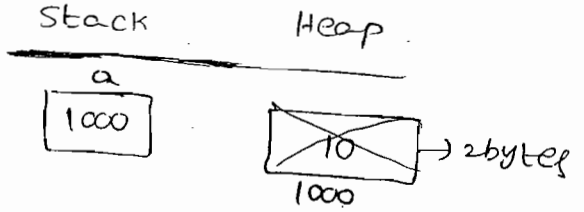
Ex :-

```
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 << "\n input the value of n";
    cin >> n;
     a = new int[n];
     cout << "\n input elements of array";
for(index=0; index<n; index++)
     cin >> a[index];
    cout << "\n Elements are \n";
     for(index=0; index<n; index++)
     cout << "\n" << 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<<"\n input name";
cin>>name;
cout<<"\n input how many subjects ?";
cin>>n;
subject = new int[n];
cout<<"\n input subjects";
for(int index=0; index<n; index++)
cin>>subject[index];
cout<<"\n Name"<<name;
cout<<"\n Marks are \n";
for(index=0; index<n; index++)
cout<<"\n"<<subject[index];
return 0;
}
```

o/p

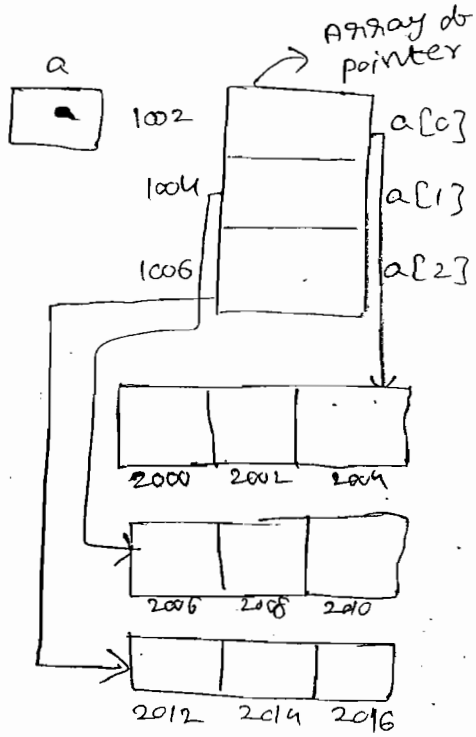
```
Input name : Bala
How many subjects ? 5
Name: Bala
marks are
80
90
70
60
85
```

creating Dynamic double dimentional array :-

1. Declare a double pointer.
2. Create an array of pointers.
3. create an array of values which hold by each pointers.

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<< "\n input how many rows";
  cin >> m;
 cout<< "\n input how many columns";
  cin>>n;
  a = new int *[m];
for(int i=0; i<m; i++)
    a[i] = new int[n];
    cout<< "\n input elements of matrix";
      for(i=0; i<m; i++)
      for(j=0; j<n; j++)
      cin >> *(*(a+i)+j);
     cout<< "\n Elements are \n";
       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 initilized 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;
clrscr();
cout << "\n" << x;
cout << "\n" << y;
cout << "\n" << z;
  z = 60;
cout << "\n" << x;
cout << "\n" << y;
cout << "\n" << z;
return 0;
}
```

o/p

10
10
10
60
60
60

Functions

@ What is Function?

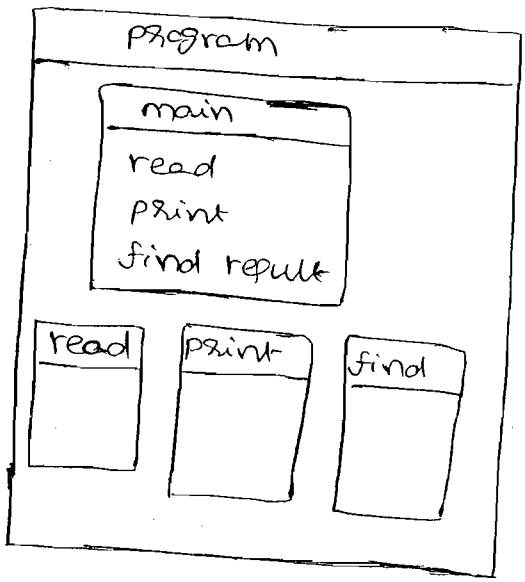
Ⓐ * A function is self contained block, which contain set of instructions

* A function is a small program within a program.
* A function is a module.
* A function is a named block.
* A function is a reusable code component.
* A function is subroutine.

Advantages of functions :-

1. Modularity :-

Dividing programming instructions according to its operations into small modules.



2. Reusability :-

Write code once and use many times.

3. Readability :-

Easy to understand application operations.

4. Efficiancy :-

It increases an Efficiancy of the program

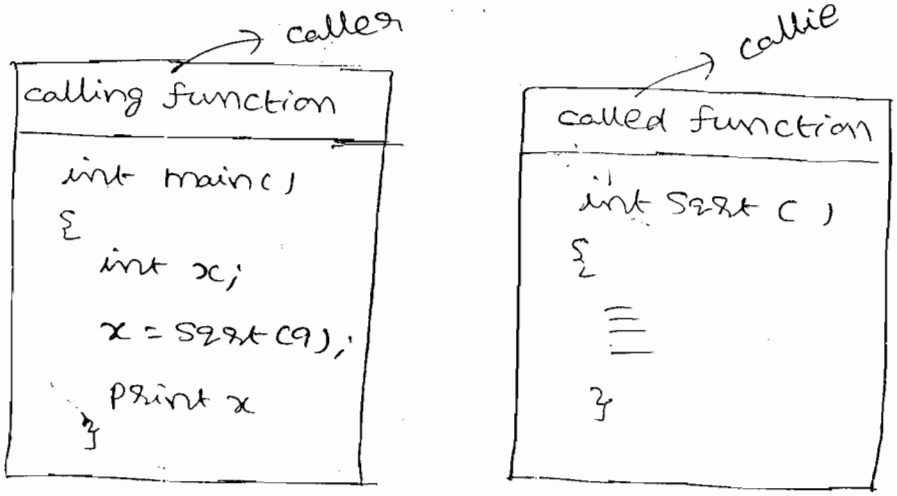
Properties or components of the Functions :-

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. (4)

(3) parameters :-

* parameters are local variables, which receive data/values from calling functions

* There are two types.

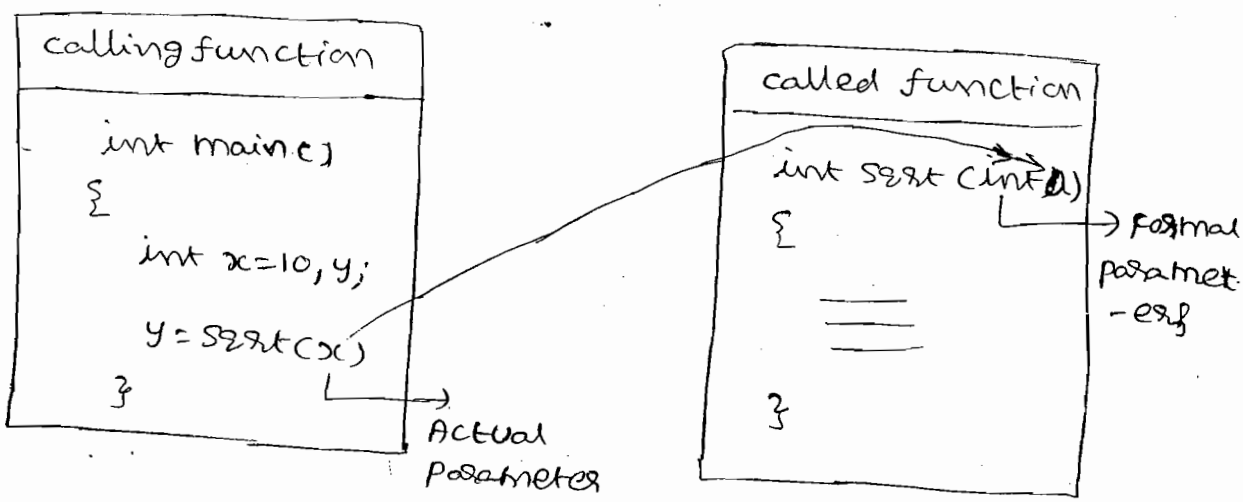
1. Actual parameters/arguments.
2. formal parameters/arguments.

1. Actual parameter :-

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

2. formal parameters :-

* 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 as 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 Functiondefinition having function header + body (contents)

```
int max(int, int);   ⟶ This is Function declaration.

int max(int x, int y)
{
}                    ⟶ 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 and with out parameters.

```
#include<iostream.h>
#include<conio.h>
void draw_line();
int main()
{
   clrscr();
   draw_line();
   cout<<"\n c++";
   draw_line();
   cout<<"\n object oriented";
   draw_line();
   return 0;
}
void draw_line()
{
    int i;
   cout<<"\n";
   for(i=1; i<=40; i++)
   cout<<"*";
}
```

o/p

****
c++
****
object oriented
****

* on invoking of a function Execution control switched from calling function to called function after processing called function, the control return back to calling 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 ()
{
   cout << "Hello";
   return;
   cout << "Bye";
}
int main ()
{
   print ();
   return 0;
}
```

O/P

Hello

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;
   }
int main()
{
  clrscr();
 int big;
  big = max(20, 40);
  cout << "\n maximum is " << big;
   return 0;
  }
```

o/p

maximum is 40

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;
    clrscr();
    cout<<"\n input a,b values";
    cin>>a>>b;
    swap(a, b);
    cout<<"\n After swaping \n";
    cout<<"\na="<<a;
    cout<<"\nb="<<b;
    return 0;
}
```

o/p

Input a,b values

10

20

After swaping

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<<"\n input a, b values";
 cin >> a >> b;
  swap (&a, &b);
  cout<<"\n After swaping \n";
  cout<<"\n a="<<a;
  cout<<"\n 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] = '\0';
}
int main()
{
char str1[10], str2[10];
  clrscr();
cout<<"\n input any string";
 cin>> str1;
 String_copy(str2, str1);
cout<<"\n String1 : "<< str1;
cout<<"\n 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?

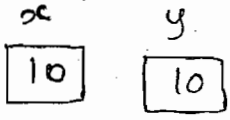
A. Deep copy involves using the contents of one object to create another instance of the same class.

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.

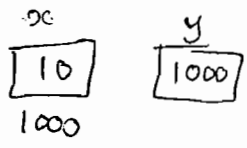
Deep copy

```
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
                              are call by reference type */
{
   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 :-

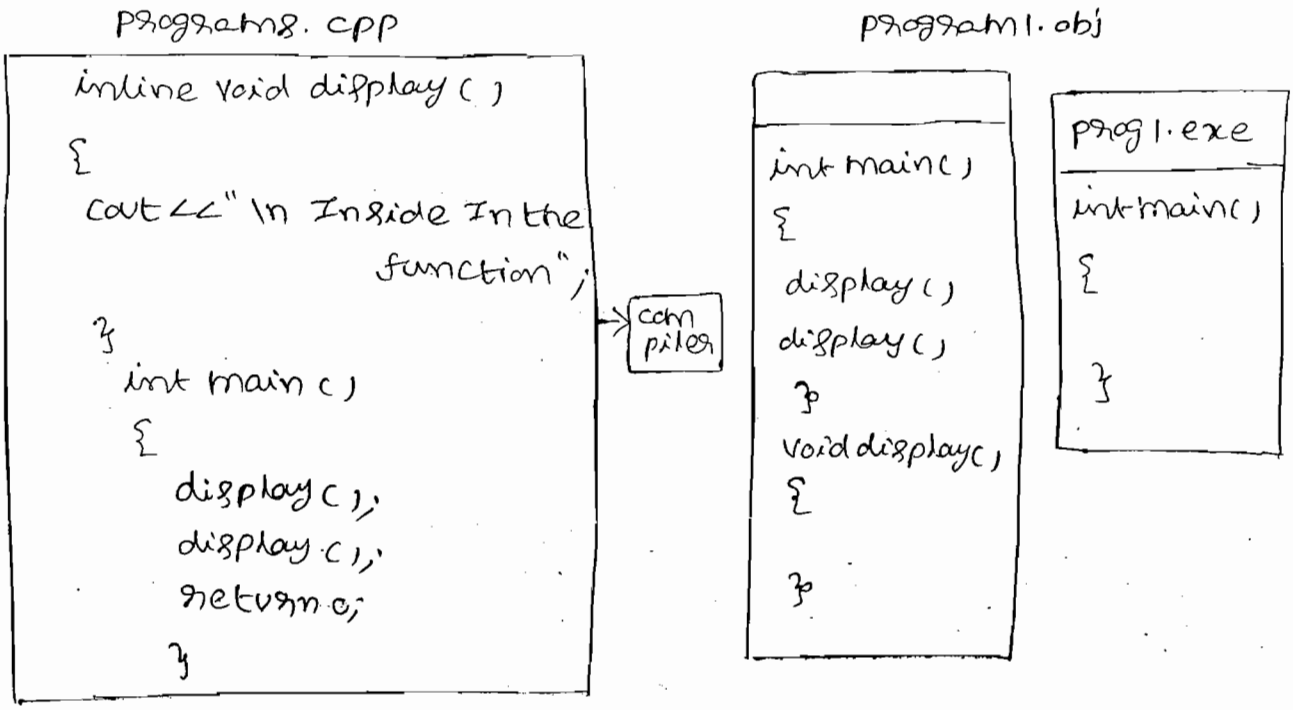
* An Inline function is declared with the key-word inline.

* 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
}
```



(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.

Explantion :

Inline-expantion could fail if the inline function contains loops, the address of an online 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 if the inline function is recursive

Ex :- program:

```
#include <iostream.h>
#include <conio.h>
inline void display()
{
 cout<< "\n Hello";
 }
 int main()
 {
  class cr();
   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 function1 (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 ( );
  cout << "x = " << x;
}
int function1 ( )
{
  int a=5;
  return a++, a++, a++;
}
```

o/p
7

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

* Defaults 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_intrest (6000, 12);
cout<<"\n simple Intrest is" << si1;
float si2 = simple_intrest (8000, 24, 2.0);
cout<<"\n simple Intrest 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, no. of parameters
  2. types of parameters.
  3. orders 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++ compilers 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 linkers 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<<"\n Area of circle"<<area1;
    cout<<"\n Area of traingle"<<area2;
      return 0;
    }
```

O/P

```
Area of circle 7.08075
Area of traingle 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 << "\n sqr of integer " << s1;
    cout << "\n 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;
```

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;
}
```

o/p

After swaping

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

* From the above program void swap (int x, int y) After including above function compiler program By compiling the program c++ compiler displays an 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.

(Q) Find out put ?

```
void function (float x)
{
 cout<<"x="<<x;
}
 int main()
 {
   function1 (10);
   function1 (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	Structure in C++
* collection of variables (or) data	* collection of variables and functions
* The contents of structure is global/public	* contents of structure is private, protected or public.
* In both the Language the structure is called users defined data type	

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

Structures in C++	class in C++
* collection of variables and functions	* collection of variables and functions
* contents of structures is private, protected and public	* contents of structure is private, protected and public
* Default content of structure is public	* Default content of class is private
* not inherited	* Inherited

* In c++ The Structures are used to build simple data types The complex data types are used to develop classes.

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

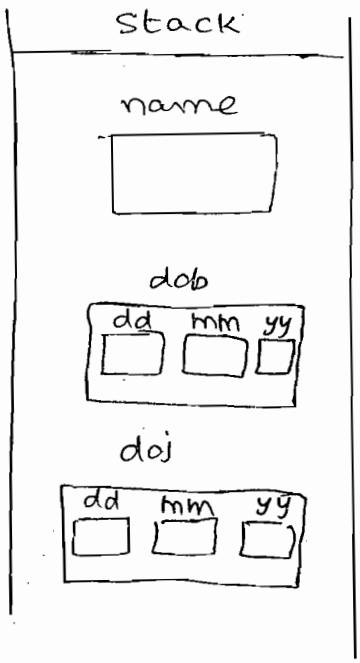
// Accessing contents of structure

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 << "\n input name",
   cin >> name;
   cout << "\n Input date of birth",
   cin >> dob.dd >> dob.mm >> dob.yy ,
   cout << " \n Input date of Joining",
  cin >> doj.dd >> doj. mm >> doj.yy;
   cout << "\n Name " << name,
   cout << "\n Date of Birth" << dob.dd << "/"
        << dob.mm << "/" << dob.yy;
  cout << "\n 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 with in the class.
* public members can be accessed from outside the class.
* The key feature do data hiding of object oriented programming is achieved by private declaration.
* The use do 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 clac
{
    int num1, num2;
};
```

```
int main ( )
{
   calc  calc1 ;
   ↓     ↓
  class  object

   calc1. num1 = 10;
   calc1. num2 = 20;   } This is ERROR

    calc calc2 ;

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

* 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 date 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;
    ↓      ↓
   class  object/variable

    calc1.add ();
     calc1. sub ();
    }
```

} member functions

* 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.

→ out side the class.
→ Inside the class.

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

Inside the class	out side the class
① Inline function is expanded during compile time.	① does not become Inline.
② decrease Efficiancy of program	② does not decrease Efficiancy of program.

program :

```
// write a program to find area of triangle
#include<iostream.h>
#include<conio.h>
class traingle
{
  float base, height;
   public:
    void read()
    {
     cout<<"\n 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<< "\n Area of triangle is " <<area1;
    return 0;
}
```

Note:-

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

Defining the members function out side the class:

The general form of members function definition,

```
Returntype classname::functionname(arguments)
{
   statements;
}
```

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 << "\n input r";
    cin >> r;
}
float circle :: find_area ()
{
    return 3.147 * r * r;
}
int main ()
{
    circle circle1;
    clrscr ();
    circle1.read_r ();
    float area = circle1.find_area ();
    cout << "\n Area of circle is " << area;
    return 0;
}
```

output

Input r 3.0

Area of circle is

member function with parameters :-

(Q) Define a way other than using the keyword inline to make a function inline?

(A) 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	method
* objects communicate by sending messages to each other	* provides response to a message
* A message is sent to invoke a method.	* It is an implementation of an operation.

program :

```
// members 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<<"\n" <<dd << "\t" <<mm <<"\t" <<yy;
  }
};
```

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

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 is called mutable object.
* An object whose state can not 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<<"\n Name "<< name;
    cout<<"\n course"<< course;
    }
    int main()
    {
        student stud1;
         clrscr();
         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;
     class2 c1;
    rect1.set_l(2.5);
    rect1.set_b(2.0);
  float area = rect1.find_area();
cout<<"\n Area is "<<area;
  return 0;
}
```

o/p

Area is 5

// 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<<"\n Input x,y values";
    cin>>x>>y;
  }
```

```
Void compare (A obj)
{
  if (x==obj.x && y==obj.y)
   cout<<"\n objects are equal";
   else
    cout<<"\n objects are not equal";
  }
};
  int main()
  {
    A obj1, obj2;
      clrscr();
      obj1.read_A();
      obj2.read_A();
      obj1.compare(obj2);
implicit ←┘                └→ Explicit
      return 0;
    }
```

---

* 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 << "\n Input complex numbers";
   cin >> real >> img;
}
Void complex :: print_complex ( )
{
  cout << "\n" << real << "\t" << img;
 }
 complex complex :: add_complex(complex c2)
 {
     complex c3;
       c3.real = real + c2.real;
       c3.img = img + c2.img;
        return c3;
    }
    int main ( )
   {
     complex compl, comp2, comp3;
        clrscr ( );
        compl.read_complex ( );
        comp2.read_complex ( );
         comp3 = compl.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<<"\n 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 <<"\t" << 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.

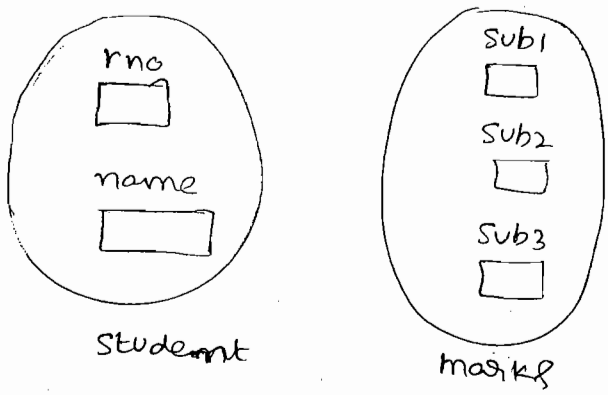
```
class A
{
  int x,y;
 public:
friend void print(A,B);
 }
```

```
class B
{
 int p,q;
 public:
friend void print(A,B);
 }
```

```
void print(A obj1, B obj2)
 {
     body of friend function
 }
```

Program:

```
#include<iostream.h>
#include<conio.h>
 class A
{
    int x,y;
    [redacted]
    friend void print(A obj);
    public:
    void print(A obj)
     {
       cout<<obj.x<<"\n"<<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<<"\n Input rollno";
    cin>> rno;
    cout<<"\n Input name";
    cin >> name;
 }
void marks :: read_marks ()
{
 cout<<"\n Input 3 subject marks";
 cin >> sub1 >> sub2 >> sub3;
}
void find_result (student s, marks m)
{
 cout<<"\n Rollno" << s. rno;
 cout<<"\n Name" << s.name;
if (m.sub1 <40 || m.sub2 <40 || m.sub3<40)
 cout<<"\n Result is fail";
 else
  cout<<"\n Result is pass";
 }
    int main ()
    {
        student stud1;
        marks final;
        clrscr ();
        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_A()
       {
        cout<<"\n Input x,y values";
        cin>>x>>y;
       }
   friend void compare(A o1, B o2);
  };
class B
{
   int p, q;
   public:
     void read_B()
   {
    cout<<"\n 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<<"\n objects are equal";
    else
    cout<<"\n objects are not equal";
}
int main()
{
    A obj1;
    B obj2;
    classes();
    obj1.read_A();
    obj2.read_B();
    compare(obj1, obj2);
    return 0;
}
```

Q. what is difference between member function and friend function ?

A.

member function	friend function
* belongs to one class/ same class	* not a member of any class.
* operates on similar type of objects	* operates on different type of objects
* It is find with object	* Not find with any object
* It is called using object name	* It 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

container class :-

An Existing class is called container class.

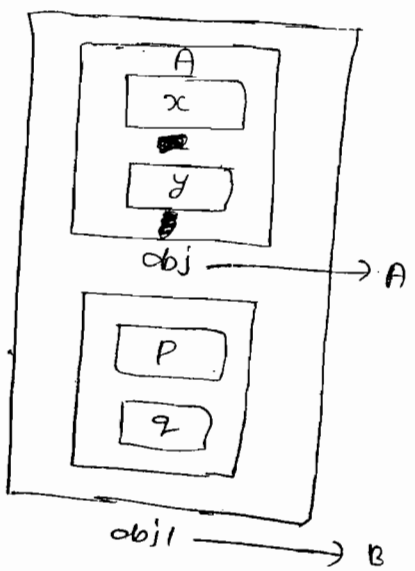
contained class :-

A class which hold object of container 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 abel to access the public members of container class.
* In order to access private members of container class contained class is declared as a friend.
* composition does not allow to extend the functionality of container 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;
   clrscr();
   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 aname[20];
  friend class book;
};
class book
{
author author_details;
 char bname[20];
 float price;
 public:
 void read_book();
 void print_book();
};
void book :: read_book()
{
  cout<<"\n Input book name";
  cin>> bname;
  cout<<"\n Input author name";
  cin >> author_details.aname;
  cout<<"\n Input price";
  cin>> price;
}
void book :: print_book()
{
  cout<<"\n Book Name"<< bname;
  cout<<"\n Author Name"<< author_details.aname;
  cout<<"\n price of book"<<
```

```
int main ()
{
    book book1;
    class2 c1;
    book1.read_book();
    book1.print_book();
    return 0;
}
```

---

*this pointer :-

→ *this holds an address of current object.

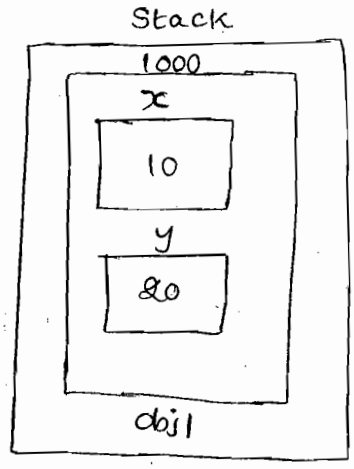
→ Every non static member function of a class having local variable *this.

→ If local variable and data member with same name, data member is refered 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<<"\n"<<this;
    }
};
int main()
{
   A obj1;
   cout<< &obj1 <<endl;
    obj1.print();
     return 0;
    }
```

Program :

```
#include<iostream.h>
#include<conio.h>
class time
{
 int hh, mm, ss;
 public:
  void set_time(int hh, int mm, int ss);
  void print_time();
 };
void time:: set_time(int hh, int mm, int ss)
{
 (*this).hh = hh;
 (*this).mm = mm;
 (*this).ss = ss;
}
void time :: print_time()
{
cout<<"\n"<<hh<<":"<< mm<<":"<< ss;
}
int main()
{
  time time_in;
   clrscr();
   time_in.set_time(10, 40, 50);
   time_in.print_time();
     return 0;
   }
```

output

10:40:50

Constructors And Destructors

Constructors :-

@ why constructors ?

(i)
```
class A
{
  int x=10;
  int y=20;
};
```
Invalid because of Data members cannot initilize with in class

(ii)
```
class A
{
  int x;
  int y;
};
int main()
{
  A obj1;
}
```
garbage values.

(iii)
```
class A
{
  public:
    int x,y;
};
int main()
{
  A obj1 = {10,20};
  A obj2 = {30,40};
}
```
Data members are public while creating values assining.

(iv)
```
class A
{
  int x,y;
};
int main()
{
  A obj1 = {10,20}; ⟶ ERROR
}
```

Constructors :-

* C++ provides is 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 destroy the objects
* constructors are used in two ways
    1. Defining Initial State of object
    2. Allocating Resources to object.
* The constructors 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 over loaded.
* 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 :-

There are three types.

(i) Default constructors.

(ii) parameterized constructor.

(iii) copy constructor.

Default constructor :-

* non parameterized constructor is called Default constructor.

(or,)

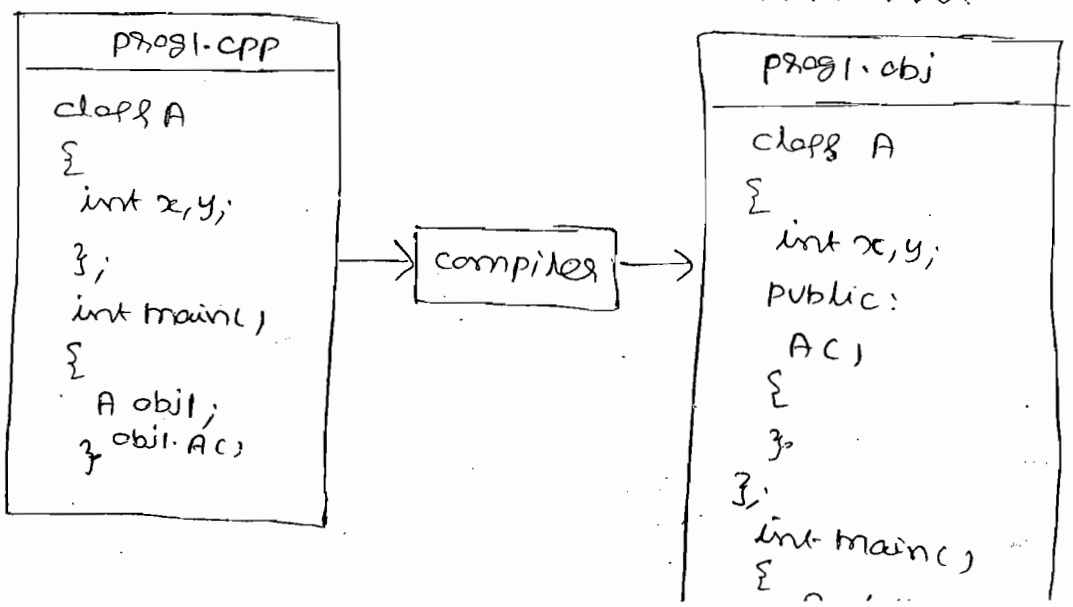
* A constructor with zero parameters is called default constructor.

* There are two types

① compiler written (compiler defined)

② user defined

→ compiler defined default constructor



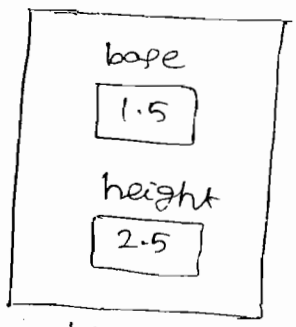
compiler defined default constructor :-

* If there is no constructor with in class compiler writes a constructor without any parameters and contents.

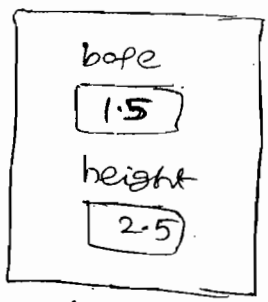
user defined default constructor :-

program :-

```
#include <iostream.h>
#include <conio.h>
 class traingle
 {
  float base, height;
   public:
     traingle ( )
     {
        base = 1.5;
        height = 2.5;
      }
 };
 int main ( )
{
 traingle traingle1 ;
 traingle traingle2;
 return 0;
 }
```



traingle1



traingle2

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 initilize object with different states.

program :

```
#include <iostream.h>
#include <conio.h>
class traingle
{
 float base, height;
 public:
  traingle (float b, float h);
  float find-area ();
 };
traingle :: traingle (float b, float h)
{
    base = b;
    height = h;
 }
 float traingle :: find-area ()
{
  return 0.5 * base * height;
 }
 int main ()
 {
```

```
Triangle Triangle1;
Triangle1.Triangle();  ⟶ Error.
```

* Because There is no default constructor with in the class.

```
Triangle Triangle1(1.5, 2.0);
Triangle Triangle2(2.5, 3.0);
clrscr();
float area1 = Triangle1.find_area();
float area2 = Triangle2.find_area();
cout<<"\n Area of Triangle 1:"<<area1;
cout<<"\n Area of Triangle 2:"<<area2;
return 0;
}
```

output

Area of Triangle1 : 1.5

Area of Triangle 2: 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;
}                          modifier
void print()
{
  cout<<x<<y;
}                          Accessor
};
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

(Q) Differences between member function and constructor?

member function	constructor
* Name can be any thing	* Name should be same as class.
* It is called any no. of Times of object	* only once
* Return type	* 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 area1 = circle1.find
```

```
float area2 = circle2.find_area();
cout<<"\n Area of circle1:"<<area1;
cout<<"\n Area of circle2:"<<area2;
return 0;
}
```

output :-

Area of circle 1 :

Area of circle 2 :

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 print_Student ( )
  {
    cout<<" \n Name:" << name;
    cout<<" \n course:" << course;
    cout<<" \n Fees:" << fees;
    }
};

int main ( )
{
  Student Stud1 (" Bala", " c" );
  Student Stud2 (" Sita", "C++", 1000 );
      class2 c );
      Stud1. print_Student ( );
      Stud2. print_Student ( );
      return 0;
    }
```

output:

```
Name: Bala
course: C
Fees: 0
Name: Sita
course: C++
Fees: 1000
```

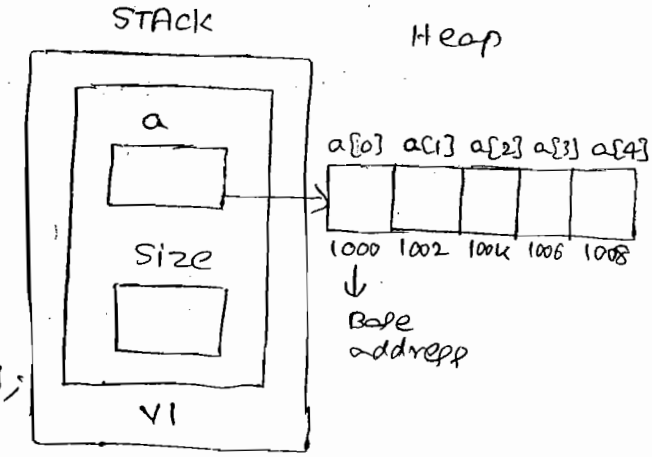
* constructor of one class is called with in another constructor of same class using the following syntax

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<<"\n Input elements";
for(int i=0; i<size; i++)
 cin>>a[i];
}
void print_elements()
{
cout<<"\n Elements are:\n";
 for(int i=0; i<size; i++)
   cout<<a[i];
  }
};
int main()
{
    clrscr();
     vector v1(5);
      v1.read_elements();
      v1.print_elements();
```



output:

Input elements
10
20
30
40
50

Elements are
10 20 30 40 50

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 inorder 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);
    clrscr();
    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 releases

Syntax:

```
~destructor_name()
{
  statements;
}
```

Program:

```
#include<iostream.h>
#include<conio.h>
class A
{
  public:
  A()
  {
   cout<<"\n Inside constructor";
  }
  ~A()
  {
   cout<<"\n Inside destructor";
  }
};
int main()
{
   A obj1;  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<<"\n Input elements:";
 for (int i=0; i<rsize; i++)
 for (int j=0; j<csize; j++)
  cin >> *(*(a+i)+j);
 }
 void print_elements()
{
cout<<"\n Elements are:\n";
 for (int i=0; i<rsize; i++)
 {
  for (int j=0; j<csize; j++)
    cout<< *(*(a+i)+j) <<" ";
```

```
    cout<<"\n";
   }
 }
 ~matrix()
 {
  for(int i=0; i<rsize; i++)
    delete *(a+i);
   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 members:

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 members is declared with in class and definition is given outside the class.

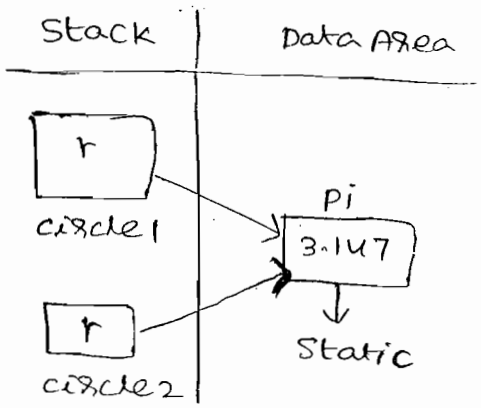
Syntax:

```
static datatype variable_name;

datatype class_name:: variable_name = [value];
```

* Static data members 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 << "\n Area of circle1: " << area1;
cout << "\n Area of circle2: " << area2;
    return 0;
}
```



* Static data members 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 obj1;
  cout<< obj1.x;
  cout<< obj1.y;
}
```

(Q) what is difference between static data members and non static data members.?

(A)

Static data members	Non Static data members
* It bind with class name and object name.	* It bind with object name
* memory is allocated on creation of object or when Accessed first time	* memory is allocated on creation of object.
* more than one object having same static Data members.	* 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.

m.t Im (Q) 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 set1(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<<"In 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 datamembers are initialize using const-ructor list/ constructor.

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;
     clrscr();
     SI.set(8000, 12);
  float SI = SI.find_interest();
cout<<"In Simple Interest is:" << SI;
  return 0;
}
```

* The value of constant data members can not 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;
  int sub1, sub2, sub3;
 public:
  void read();
 {
 cout<<"\n Input rollno";
 cin >> rno;
 cout<<"\n Input 3 subjects";
 cin>> sub1 >> sub2 >> sub3;
 }
 void display() const
{
 cout<<"\n Rollno" <<rno;
 cout<<"\n Total "<< sub1+sub2+sub3;
 cout<<"\n 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 userdefined 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
}
return type
```

another syntax :

```
operator <operator symbol> (<parameters>)
{
   <statements>;
}
```

The keyword "operator", followed by an operator symbol, defines a new (overloaded) action of the given operator.

operator is defines 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.

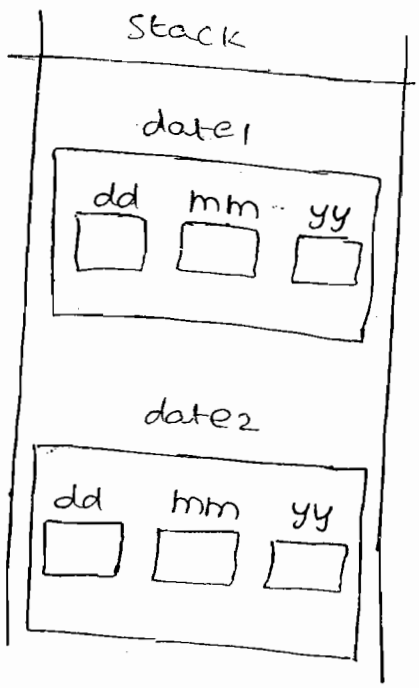
we can not change the basic meaning of an operator.

operators can not be overridden.

when binary operators overloaded Thrugh 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<<"\n Input date1:";
cin>>date1.dd>>date1.mm>>date1.yy;
cout<<"\n Input date2:";
cin>>date2.dd>>date2.mm>>date2.yy;
 if(date1 == date2)
 cout<<"\n Dates are equal";
 else
 cout<<"\n Dates are not equal";
   return 0;
 }
```



output

Input date1: 2 7 2010

They follow the syntax & 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;
    clrscr();
  cout<< "\n Input String1:";
  cin >> str1.str;
  cout<< "\n Input String2:";
   cin>> str2.str;
      str3 = str1+str2;
  cout<< "\n After concat " << str3.str;
    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(strcmp(s1.str, s2.str) == 0)
  return 1;
  else
   return 0;
}
```

```
int main()
{
 String str1, str2;
   clrscr();
 cout<<"\n Input String1:";
 cin>> str1.str;
 cout<<"\n Input String2:";
 cin>> str2.str;
 if(str1==str2)
 cout<<"\n Strings are equal";
 else
 cout<<"\n 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<<"\n Input date:";
    cin>> dd >> mm >> yy
```

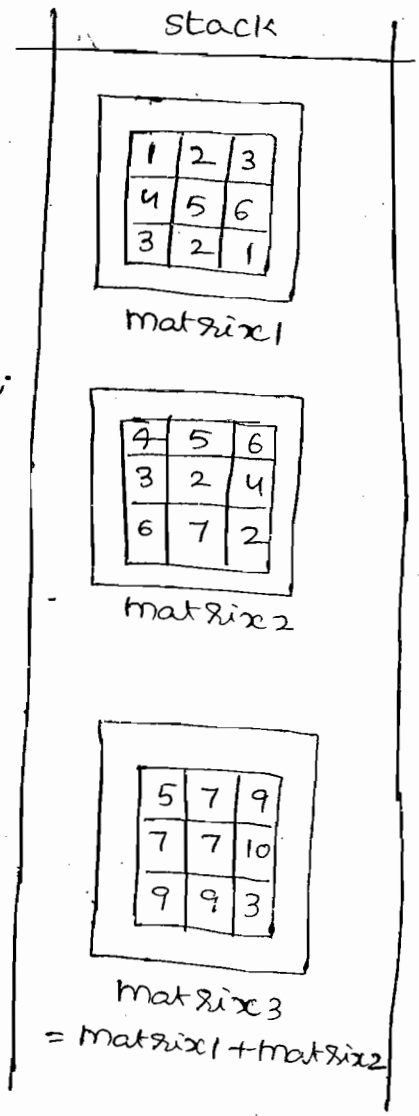
```
int operator ==(date d)
{
if(dd==d.dd && mm==d.mm && yy==d.yy)
return 1;
else
return 0;
}
};
int main()
{
date date1, date2;
clrscr();
cout<<"\n Input date1:";
date1.dd
date1.read_date();
cout<<"\n Input date2:";
date2.read_date();
if(date1==date2)
cout<<"\n Equal";
else
cout<<"\n Not equal";
return 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<<"\n 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<<"\n 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;
    clrscr();
matrix1.read_matrix();
matrix2.read_matrix();
matrix3 = matrix1 + matrix2;
matrix3.print_matrix();
    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

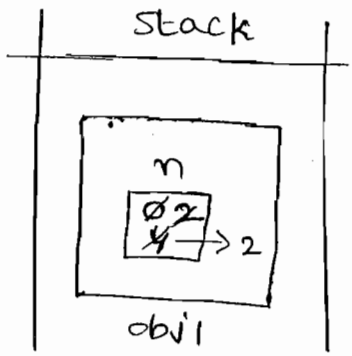
Elements are:

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 functions :-

* 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 can not be overloaded using friend.
* The operators that can not be overloaded using friend function.

Assignment operator =

Function call operator ( )

Subscripting operator [ ]

class members access operator →

members functions can be used to overload them.

Syntax:

```
friend return-type operator operator symbol(parameters);

return-type operator operator-symbol(parameters)
{
    Statements;
}
```

* The definition of friend operator is given within class or out side 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<<"\n Input x,y values";
 cin>>x>>y;
 }
friend int operator ==(A,B);
};
class B
{
 int p,q;
 public:
 void read-b()
 {
  cout<<"\n 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;
   clrscr ( );
   obj1.read_ac );
   obj2.read_b( );
 if (obj1 == obj2) // == (obj1, obj2)
 cout<<"\n objects are equal ";
 else
 cout<<"\n 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 operates on predefined datatypes does not operates on user defined data type.

* Inorder to operate on user defined types these operators are overloaded.

Insertion (<<), Extraction (>>)

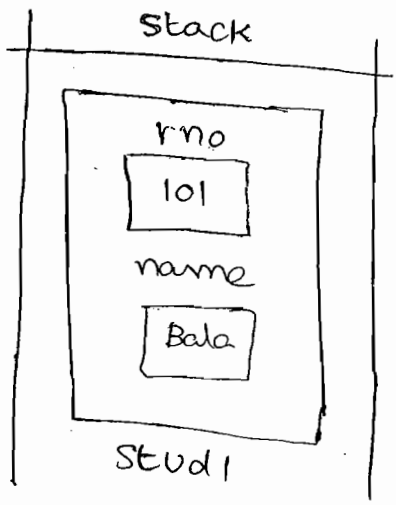
program :

```
#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;
  clrscr ( );
 cout<< "\n Input student details :";
  cin >> stud1;
 cout<< "\n 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 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 :-

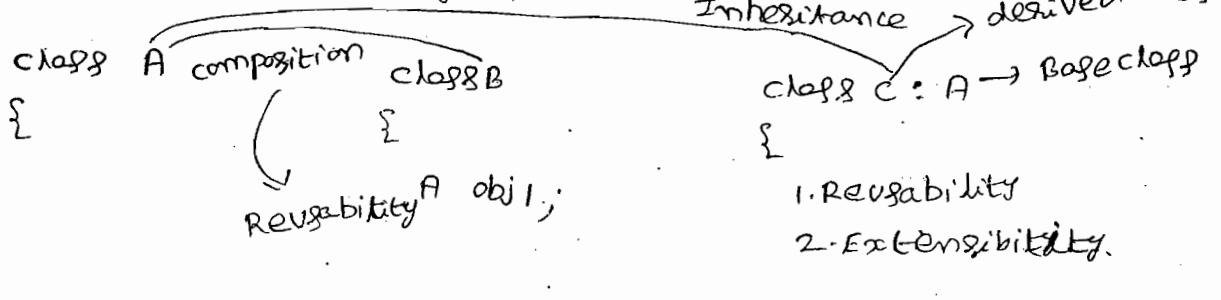
A derived class is defined by specifying its relationship with the base class.

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.



Program : Example for composition

```
class book
{
  public:
  char bname[20];
  char aname[20];
};
class publishes
{
  float price;
   book b;
  public:
   void read_details()
  {
    cout<<"\n Input book Name ";
     cin >> b.bname;
    cout<<"\n 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.read_details();
    publisher1.print_details();
}
```

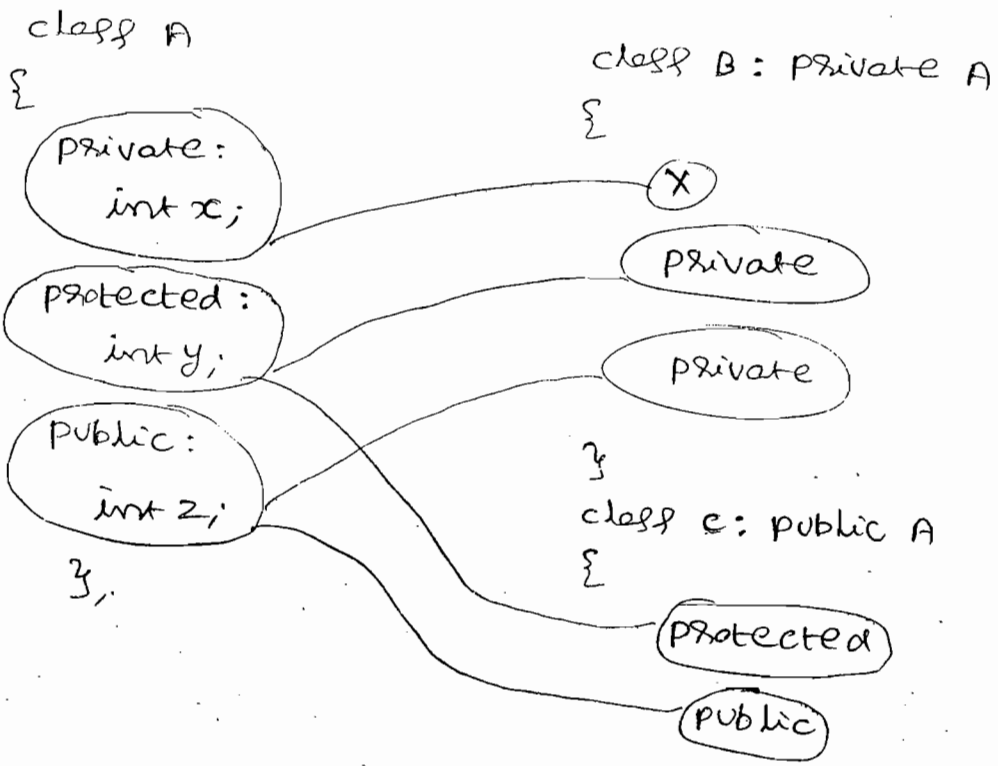
```
class derived_class : visibility mode base class,...
{

}
```

→ It means Inheritance operator

It means visibility/Access of base class members with in 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 members 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 can not access the out side 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.

Ex :- int x, y, P, q

Size = 8 bytes

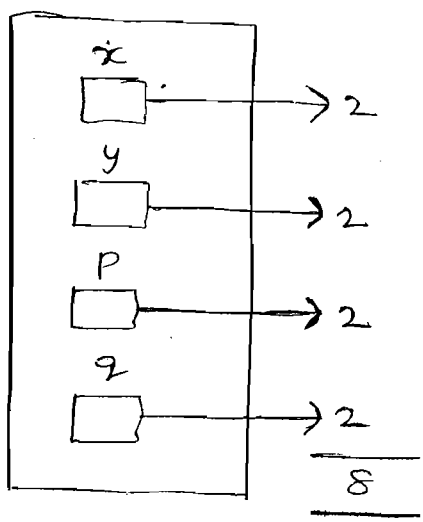
x → 2 bytes
y → 2 bytes
P → 2 bytes
q → 2 bytes

Types of Inheritances :- C++ supports 5 types of Inheritances. They are:

1. Single level Inheritance.
2. multi-level Inheritance.
3. multiple Inheritance.
4. Hierarchical Inheritance.
5. Hybrid Inheritance (or) multipath Inheritance.

What happen 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 objects 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:

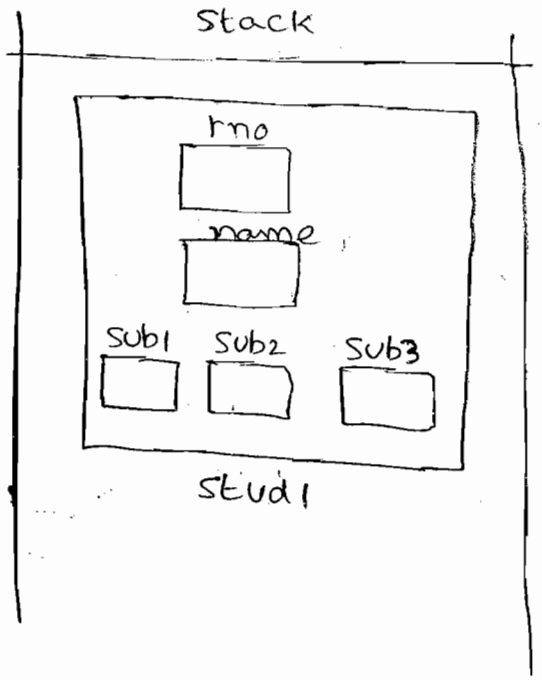
```
class base
{
   ----
   ----
};
class derived : public base
{
  ----
};
```

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<<"\n Input rollNo:";
cin>> rno;
cout<<"\n Input name:";
 cin >> name;
}
Void print-student()
{
 cout<<"\n RollNo"<< rno;
 cout <<"\n Name"<< name;
 }
};
class marks : public student
{
 int sub1, sub2, sub3;
 public:
  Void read-marks()
  {
   cout<<"\n Input 3 subjects marks:";
    cin>> sub1 >> sub2>> sub3;
  }
  Void find-result()
  {
  if( sub1<40 || sub2<40 || sub3<40)
  cout<<"\n Results is Fail";
  else
  cout<<"\n Result is pass";
  }
};
```

```
int main ( )
{
  marks stud1;
  clrscr ( );
stud1.read_student ( );
stud1.read_marks ( );
stud1.print_student ( );
stud1.find_result ( );
  return 0;
}
```



output :

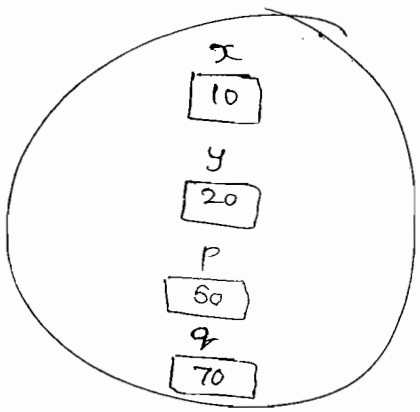
```
Input rollNo: 37
Input name: Bala
Input 3 subject marks: 90 95 85
   RollNo: 37
   Name: Bala
   Result is pass

Input rollno : 50
Input name: Lalitha
Input 3 subject marks: 95 80 35
      RollNo: 50
      Name : Lalitha
      Result is Fail
```

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;
}
```



compiler →

```
class B : public A
{
  int p, q;
  public:
   B ( ),
  {
    A::A( );
     p=60;
     q=70;
  }
};
```

program :

```
#include <iostream.h>
#include <conio.h>
 class A
{
   int x, y;
   public:
    A ( )
    {
       x=10;
       y=20;
    }
    Void print-a ( )
   {
     cout<< "\n x=" << x;
     cout<< "\n y=" << y;
    }
};

 class B : public A
{
   int p, q;
    public:
     B ( )
     {
       p = 60;
       q =70;
     }
     Void print-b ( )
     {
       cout<< "\n p=" << p;
       cout<< "\n q=" << q;
     }
};
```

output :

x=10

y=20

p=60

q=70

```
int main()
{
    B obj1;
    classA();
    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 with in base class when If any other constructor is define.

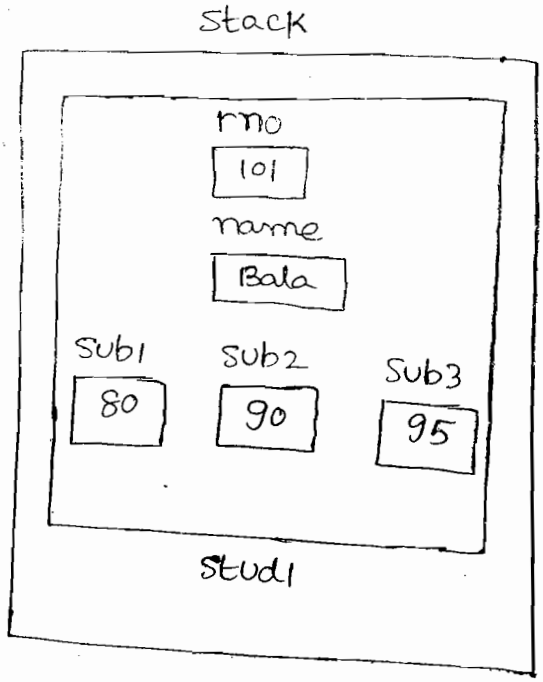
program:

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 << "\n Roll No: " << rno;
cout << "\n 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 << "\n Subject1: " << sub1;
     cout << "\n Subject2: " << sub2;
     cout << "\n Subject3: " << sub3;
     }
};
```

```
int main( )
{
  marks stud1(101, "Bala", 80,
                          90 95),
    class2 ( );
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 ";
  }
};
```

→ compiler →

```
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 ( )
{
    clrscr ( );
     B obj1;
    return 0;
 }
```

* In the above program before deleting of object B destructor of B is called which calls the destructor of A class

* Destructer can be called Explicitly using class name

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:

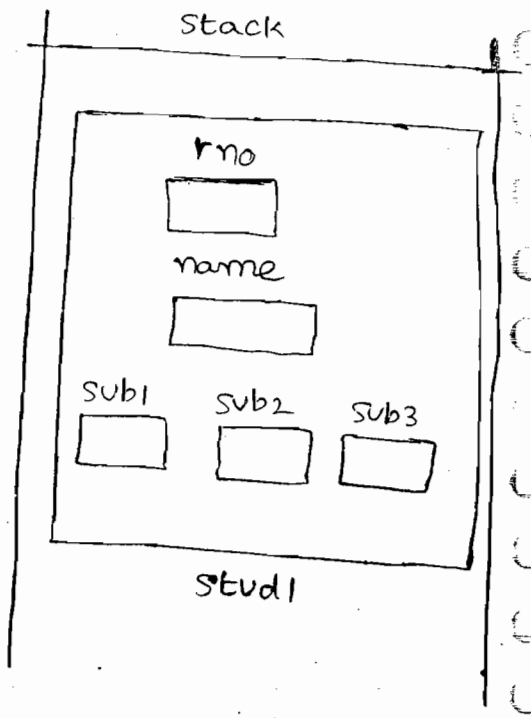
```
class A
{
 protected:
   int x, y;
 Private:
   int z;
};
```

```
class B : public A
{
  public:
   void fun1()
   {
    cout<<x<<y;
    cout<<z; ⟶ not Accessed
   }
};
```

Program:

```
#include<iostream.h>
#include<conio.h>
class student
{
   protected:
     int rno;
     char name[20];
     public:
     void read_student()
     {
       cout<<"\n Input_rollno:";
       cin >> rno;
       cout<<"\n Input name:";
       cin>> name;
     }
```

```
class marks : public student
{
    int sub1, sub2, sub3;
     public:
     void read_marks()
     {
      cout<<"\n Input 3 subject marks:";
       cin>> sub1 >> sub2 >> sub3;
     }
    void find_result()
    {
      cout<<"\n RollNo: "<<rno;
       cout<<"\n Name: "<<name;
    if(sub1<40 || sub2<40 || sub3<40)
     cout<<"\n Result is Fail";
      else
     cout<<"\n Result is pass";
    }
 };
int main()
{
   marks stud1;
  clrscr();
   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.

Exm:

```
class A
{
  int x,y;
};
```

```
int main()
{
  A *P;        → Not object. It is a pointer which hold Address of object.
  P = new A;
}
```

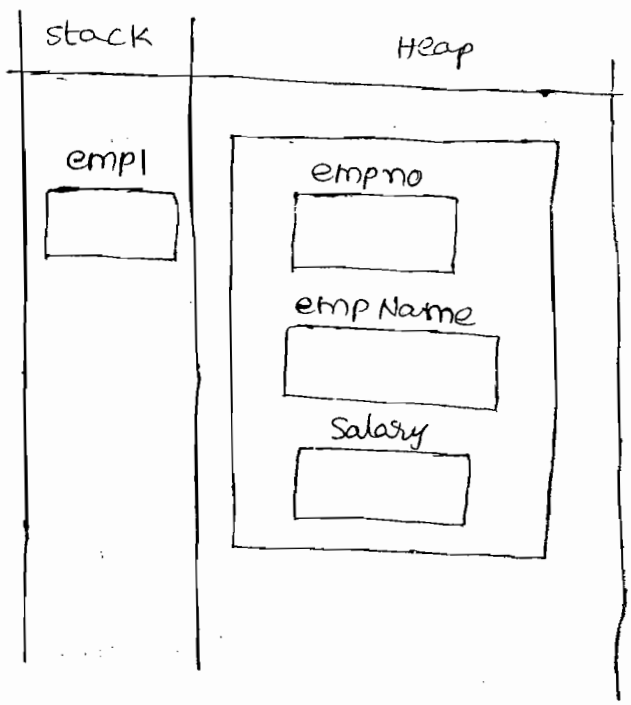
note :

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<<"\n Input employee no:";
     cin>> empno;
     cout<<" \n Input employee Name:";
      cin >> name;
     cout<<"\n Input employee Salary :";
      cin >> Salary;
     }
     Void print_Employee ()
     {
      cout<<"\n Employee No: "<<empno;
      cout<<" \n Employee Name:"<< name;
      cout<<" \n Employee salary :"<< Salary;
       }
     };
  int main ()
 {
  employee *empl;
    clrscr ();
 empl = new employee ();
(*empl). read_employee ();
(*empl). print_employee ();
    return 0;
 }
```



output:

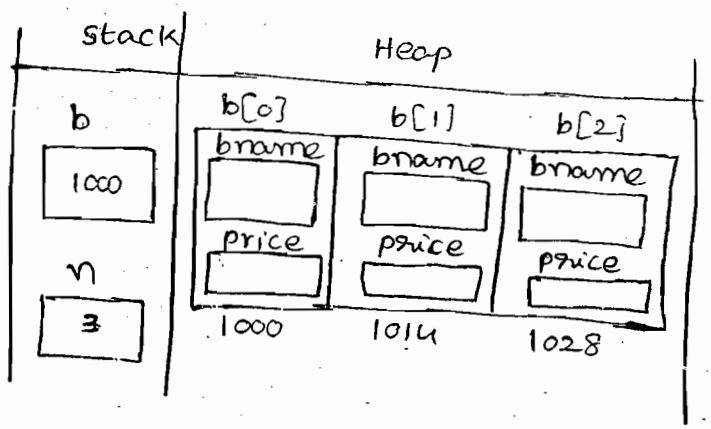
```
Input employee no : 101
Input employee name : Bala
Input employee Salary : 10,000

    Employee No: 101
    Employee Name: Bala
```

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()
 {
 cout<<"\n Input book name:";
 cin>>bname;
 cout<<" \n Input price:";
 cin>> price;
 }
 void print_book()
 {
 cout<< endl << bname<< "\t" << price;
 }
};
int main()
{
  book *b;
   int n,i;
   clrscr();
 cout<<" \n 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 dynamic derived class object

```
#include <iostream.h>
#include <conio.h>
class A
{
  public:
   void function1()
   {
    cout<<"\n Inside function1 of class A";
   }
};
```

```
class B : public A
{
  public:
   void function2()
   {
    cout<<"\n Inside function2 of class B";
   }
};
int main()
{
  B *obj;
  clrscr();
   obj = new B;
   (*obj).function1();
   (*obj).function2();
      return 0;
  }
```

---

member function overriding :

* Redefining of base class member function with in derived class is called function overriding.
* In function overriding derived class function is defined with same signature of base class.
* function overriding to allows to extend the functionality of base class with in derived class.

Ex:

```
class A
{
  public:                    ⟵ This is function overriding
   void function1()
   {
```

```
    cout<<"\n Hello";
   }
 };
class B : public A
{
 public:                          ------> This is overridden function
 void function1()
 {
  cout<<"Hello";
  cout<<"\n Bye";
 }
};
int main()
{
    B obj1;
    clrscr();
    obj1.function1();
     return 0;
 }
```

* In function overriding the functionality of base class is hidden with in derived class.

* Base class function can't call out side 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 rn;
```

```
    public:
     void read ()
    {
     cout<<"\n Input rollno:";
     cin>>rno;
     cout<<"\n Input name:";
     cin>>name;
    }
     void print()
    {
     cout<<"\n Roll no:"<<rno;
     cout<<"\n Name:"<<name;
    }
  };
class marks : public student
{
  int sub1, sub2, sub3;
   public:
    void read()
    {
     student::read();
     cout<<"\n Input 3 subjects:";
     cin>> sub1 >> sub2 >> sub3;
    }
    void print()
    {
      student::print();
      if(sub1<40 || sub2<40 || sub3<40)
       cout<<"\n Result fail:";
       else
       cout<<"\n Result pass";
      }
  };
```

```
int main ( )
{
  marks stud1;
    class1 ( );
  stud1.read ( );
  stud1.print ( );
  return 0;
}
```

Static Binding :

In function overloading or operator overload-ing 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 of the selection of the function is done dynamically at runtime.

Dynamic binding requires use of pointers to objects.

The ~~data~~ 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 ";
 }
};
```

```
int main()
{
 B *child;
 child = new B;
(*child).fun1();
(*child).fun2();
 A *parent;
 parent = new A;
(*parent).fun1();
 return 0;
}
```

Virtual functions:

To invoke the derived class member function by base 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

Rules for virtual functions :-

* The virtual functions must be members of some class
* They can not be static members.
* They 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 function1 ( )
   {
    cout<<"\n Inside function1 of class A";
    }
},
```

```
class B : public A
{
  public:
   void function1 ( )
  {
  cout << "\n Inside function1 of class B";
   }
   void function2 ( )
  {
  cout<< "\n Inside function2 of class B";
  }
};
int main ( )
{
  A *p;
   clrscr ( );
   p = new A ;
 (*p). function1 ( );
   delete p;
   p = new B;
 (*p). function1 ( );
 (*p). function2 ( );  // error
   return 0;
}
```

output :

Q) 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
{
  protected:
  float dim1, dim2;
```

```
public:
    void read ()
    {
    cout<<" In Input 2dim";
    cin>> dim1 >> dim2;
    }
virtual float find_area () = 0;
};
class traingle : public shape
{
public:
float find_area ()
{
  return  0.5*dim1*dim2;
}
};
class rectangle : public shape
{
  public:
 float find_area()
 {
   return  dim1*dim2;
 }
};
int main()
{
   traingle traingle1;
       clrscr();
       traingle1.read();
      float a1 = traingle1.find_area();
        cout<<" In Area of traingle is "<< a1;
```

```
    rectangle rectangle1;
    rectangle1.read();
  float a2 = rectangle1.find area();
cout<<"\n 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).find_area();
        delete shape1;
        shape1 = new rectangle;
        (*shape1).read();
        (*shape1).find area();
    }
```

* PURE virtual function 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 salaried_employee : public employee
{
   float salary;
    public:
    void read ( )
   {
   cout<<" In Input employee no: ";
   cin >> empno;
   cout<< "\n Input employee name: ";
   cin>> name;
   cout<<"\n Input salary ";
    cin>> salary;
   }
   void print ( )
   {
    cout<< " \n Employee No: " << empno;
    cout<<" \n Employee Name: " << name;
    cout<<"\n Salary " << salary;
     }
  2.
```

```
class daily_workers: public employee
{
    float wage;
     public:
     void read ()
    {
    cout<<" In Input employeeno:";
     cin>> empno;
    cout<<" In Input employee name:";
     cin>> name;
    cout<<" In Input daily wage :";
     cin>> wage ;
  }
  void print ()
 {
  cout<<" In EmployeeNo: "<< empno;
  cout<<" In Employee Name:" << name;
  cout<<" In Employee wage:" <<wage;
  }
};
int main ()
{
    employee *emp;
      clrscr ();
      emp = new salaried_employee;
    (*emp). read ();
    (*emp). print ();
       delete emp;
```

```
emp = new daily_workers;
(*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 specifier

Program: // Hybrid inheritance

```
#include <iostream.h>
#include <conio.h>
class A
{
   public:
   void fun1 ()
   {
   cout<<"\n Inside function1 of class A";
   }
};
class B: virtual public A
{
   public:
   void fun2()
   {
  cout<<"\n Inside function2 of class B";
   }
};
class C: public virtual A
{
  public:
  void fun3 ()
 {
  cout <<"\n Inside function3 of class C";
  }
};
```

```
class D: public B, public C
{
 public:
   void fun4 ( )
   {
   cout<<"\n Inside function4 of class D";
   }
};
int main ( )
{
   D obj1;
   clrscr ( );
   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 constructors.

* compiler bind or invoke members using pointer type. In order to invoke based on object type we declared base class members as virtual.

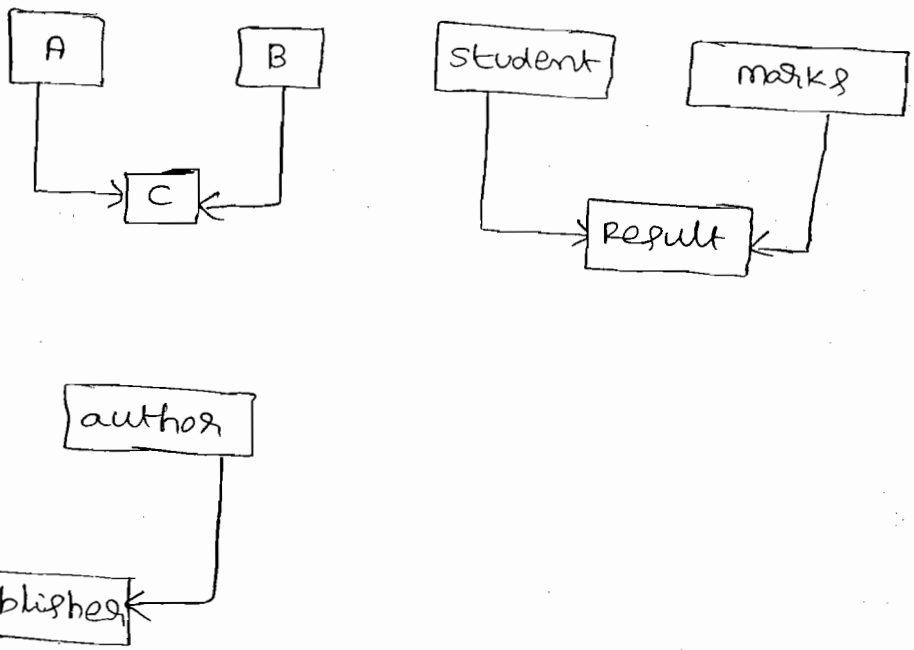
multiple Inheritance :

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
{

};
```

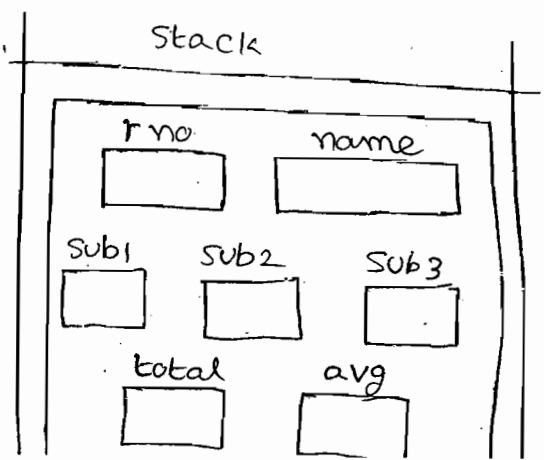
Examples :



Program :

```
// multiple Inheritance

#include<iostream.h>
#include<conio.h>
class student
{
   int rno;
   char name[20];
   public:
   void read_student()
  {
  cout<<"\n Input roll no";
   cin>>rno;
  cout<<"\n Input name";
   cin>>name;
   }
  void print_student()
 {
  cout<<"\n RollNo:"<<rno;
  cout<<"\n Name:"<<name;
  }
};
 class marks
{   protected:
   int sub1, sub2, sub3;
    public:
   void read_marks()
  {
   cout<<"\n Input 3 subject marks";
    cin>>sub1>>sub2>>sub3;
   }
```

```
    void print_marks ()
    {
      cout<<"\nSubject1:"<< sub1;
      cout<<"\n Subject2:" << sub2;
      cout<<"\n Subject3:" << sub3;
      }
   };
class Result : public student, public marks
{
    int total;
    float Avg;
    public:
    void find_result ()
    {
      total = sub1 +sub2+sub3;
       avg = total/30;
       cout<<"\n Total:" << total;
       cout<<"\n Avg :" << avg;
   if ( sub1<40 || sub2<40 || sub3 < 40)
   cout << "\n Result Fail ";
   else
   cout<<"\n Result pass ";
    }
};
```



```
int main ( )
{
    result stud1;
      clrscr ( );
    stud1. read - student ( );
    stud1. read - marks ( );
    stud1. print - student ( );
    stud1. print - marks ( );
    stud1. find - result ( );
     return 0;
}
```

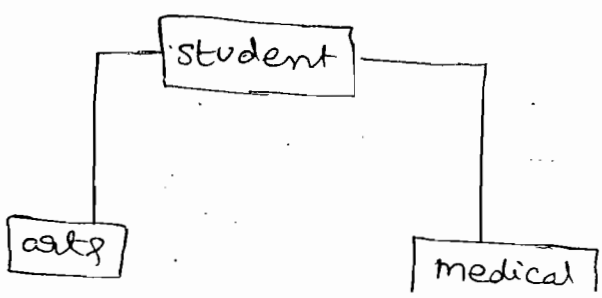
Hierarchical inheritance :

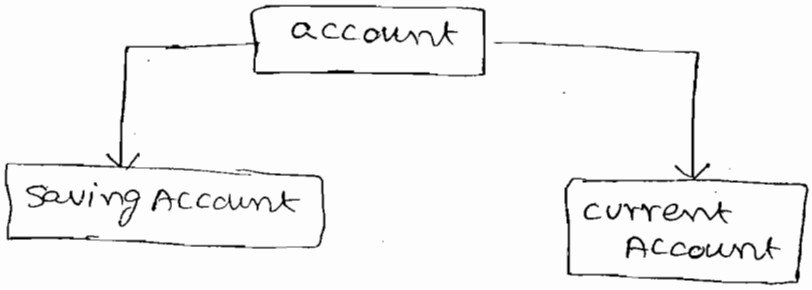
If class is inherited by more than one class it is called hierarchical inheritance.

Exm :

```
class student
{
};
class arts : 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 accant
{
  public:
    void read-account()
  {
  cout<<"\n Input account-no:";
  cin >> accno;
  cout<<"\n Input customer name:";
  cin>> cname;
```

```
        cout<<"\n Input balance :";
        cin>> balance;
    }
    Void print-account ()
{
    cout<<"\n AccountNo :" <<accno;
cout<<"\n Customer Name:" <<cname;
if (balance <1000)
cout<<"\n Invalid balance " <<balance;
 else
cout<<"\n balance " <<balance;
}
};

class currentaccount : public account
{
    public :
    Void read-account ()
    {
      cout<<"\n Input account no:";
       cin>> accno;
       cout<<"\n Input customer Name :";
        cin>> cname;
       cout<<"\n Input balance :";
        cin>> balance;
       }
       Void print-account ()
       {
         cout<<"\n Account No :" <<accno;
```

```
    cout<<"\n Customer Name: "<<cname;
     if (balance <2000)
    cout<<"\n Invalid balance "<< balance;
     else
     cout<<"\n balance "<<balance;
   }
};
 int main ()
{
    account *acc;
      clrscr();
     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

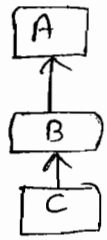
Ex :

```
class student
{
};
class test : public student
{
};
class sports : public student
{
};
class result : public test, public sports,
```

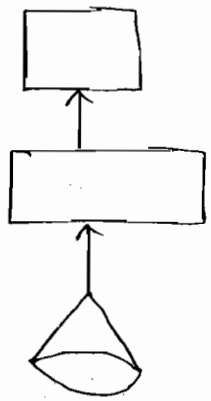
This is also called as multi path inheritance as the class student inherited via test and also via sports.

Multilevel Inheritance :

When the derived class is Inherited from the other derived class, It is called multilevel Inheritance. The typical block diagram of multilevel Inheritance like is like this



Ex :



The general format of multilevel 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 area()
    {
     cout<< l*l;
    }
};
class Rectangle : public Square
{
    protected :
          int b;
     public :
          void Accept2()
     {
        Accept1();
        cout<< "Enter breadth";
        cin>> b;
     }
```

```
Void Area 2 ( )
 {
   cout<< l*b;
  }
};

class Pyramid : public Rectangle
 {
    private :
            int h;
     public :
            void Access 3 ( )
            {
              Access 2 ( ),
            cout<< "Enter height ";
             cin>> h,
             }
           Void area 3 ( )
          {
           cout<< 1/3 *b*h;
          }
 };
 void main ( )
 {
     Pyramid obj;
       obj. Access Pyramid ( ),
        obj. area ( ),
          getch ( );
    }
```

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 constructs 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 function1 < T1 x, T1 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
}
```

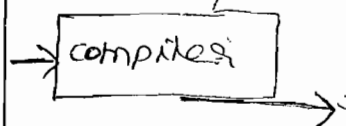
TType is a place holder name for a data type used by the function. This name may be used with in a function definition. The compiler automatically replaces it with an actual data type when it creates a specific version of the function.

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;
}
```




```
int max (int x, int y)
{
   if (x>y)
   return x;
   else
   return y;
}
float max (float x, float y)
{
   if (x>y)
   return x;
   else
   return y;
}
```

Program:

```
#include <iostream.h>
#include <conio.h>
template < class T1 , class T2>
Void function1 (T1 x, T2 y)
{
 cout<< "\n x =" << x;
```

```
int main ()
{
  class c;
  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 X>
 X max (X a, X 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()
{
    classA();
  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<< "\n" << p;
  cout<<"\n" << q;
  cout<<"\n" << m;
  cout<< "\n" << 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
  {
        ----
        ----
```

Ttype 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.

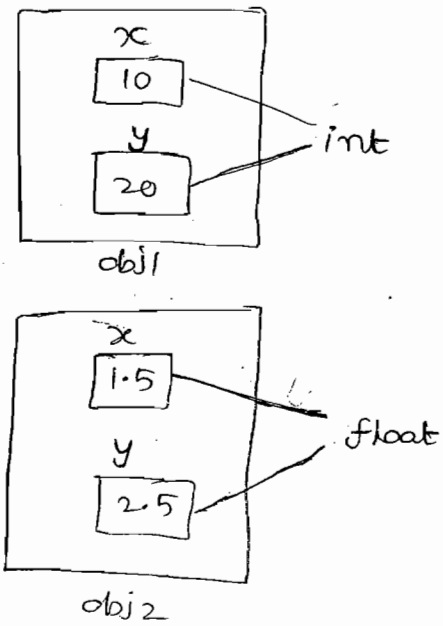
class template:

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;
   }
```

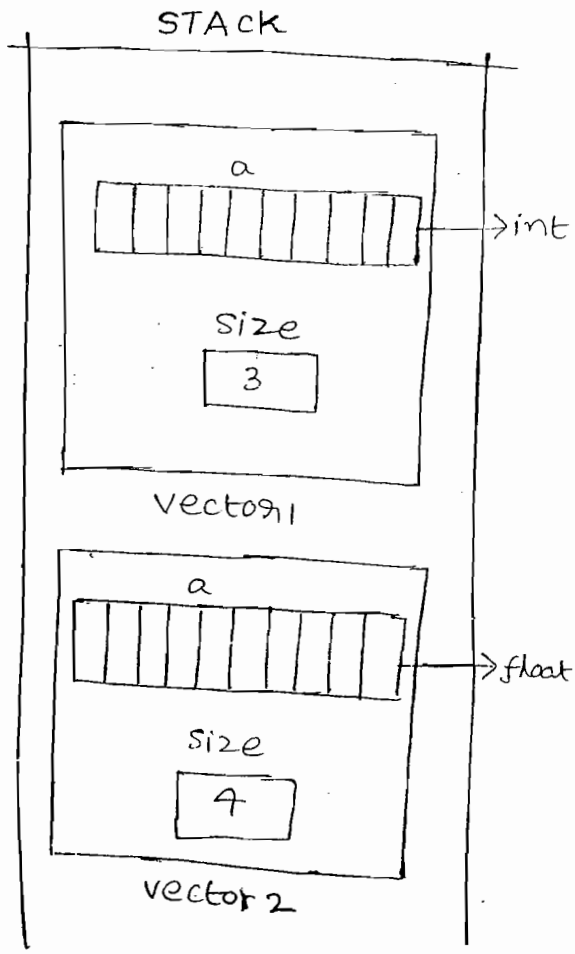
```
Void set (T x, T y)
{
 (*this).x = x;
 (*this).y = y;
 }
};
int main()
{
  A <int> obj1;
  A <float> obj2;
    obj1.set(10,20);
    obj2.set(1.5,2.5);
     obj1.print();
     obj2.print();
     return 0;
}
```



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<<"\n Input Elements of vector:";
     for(int i=0; i<size; i++)
      cin >> a[i];
     }
    void print_elements() {
     cout<<"\n Elements are:\n";
      for(int i=0; i<size; i++)
       cout<<"\n"<< a[i];
       }
    };
int main() {
[redacted]
vector<int> vector1(3);
vector<float> vector2(4);
     closer();
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<< "\n" << x;
 cout<< "\n" << y;
}
int main()
{
 print (10,20);
 print (10,1.5);
  return 0;
}
```

program :

A class can have multiple data types.

```
#include <iostream.h>
#include <conio.h>
template <class T1, class T2>
 class A
 {
    T1 x;
    T2 y;
    public:
    Void read()
   {
    cout<< "\n Input x,y values :";
      cin>> x >> y;
    }
```

```
Void read ( )
{
  cout<<"\n Input x, y values : ";
   cin >> x >>y;
 }
```

```
Void print ( )
{
   cout<<"\n" << x;
   cout<<"\n" << y;
  }
};
int main ( )
{
   A < int, float> obj1;
   A < float, float> obj2;
      clrscr ( );
  cout<<"\n Size of object1 " << sizeof(obj1);
cout<<"\n Size of object 2" << sizeof(obj2);
        obj1. read ( );
        obj1. print ( );
        obj2. read ( );
        obj2. print ( );
       return 0;
  }
```

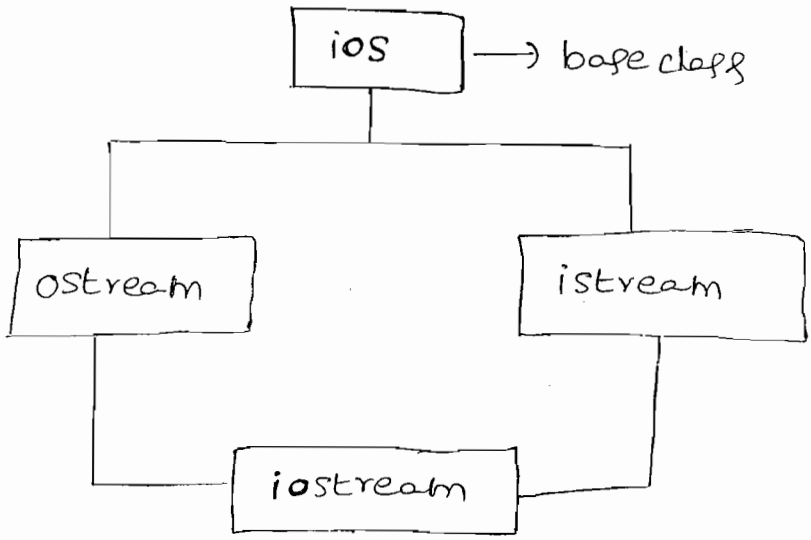
Streams and Files

* 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 character. get( ) can be used in two ways.

cin.get(ch); // get a character from keyboard and assigns 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<<"\n No of characters:"<<cc;
  cout<<"\n No of words:"<<wc;
  cout<<"\n 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.

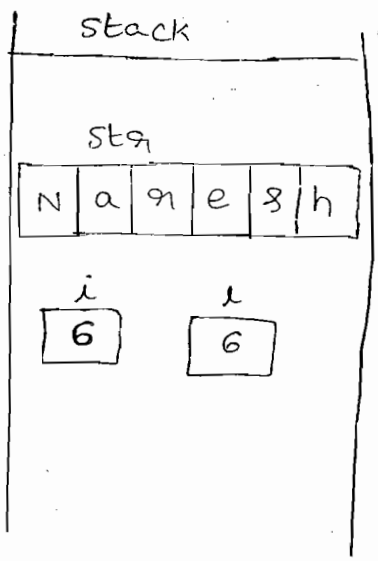
write( ) function displays the entire line.

cout.write(line, size);.

line represents the name of the string 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 << "\n Input any string:";
  cin >> str;
    l = strlen(str);
 for(i=1; i<=l; i++)
 {
   cout.write(str, i);
   cout << "\n";
  }
   return 0;
}
```

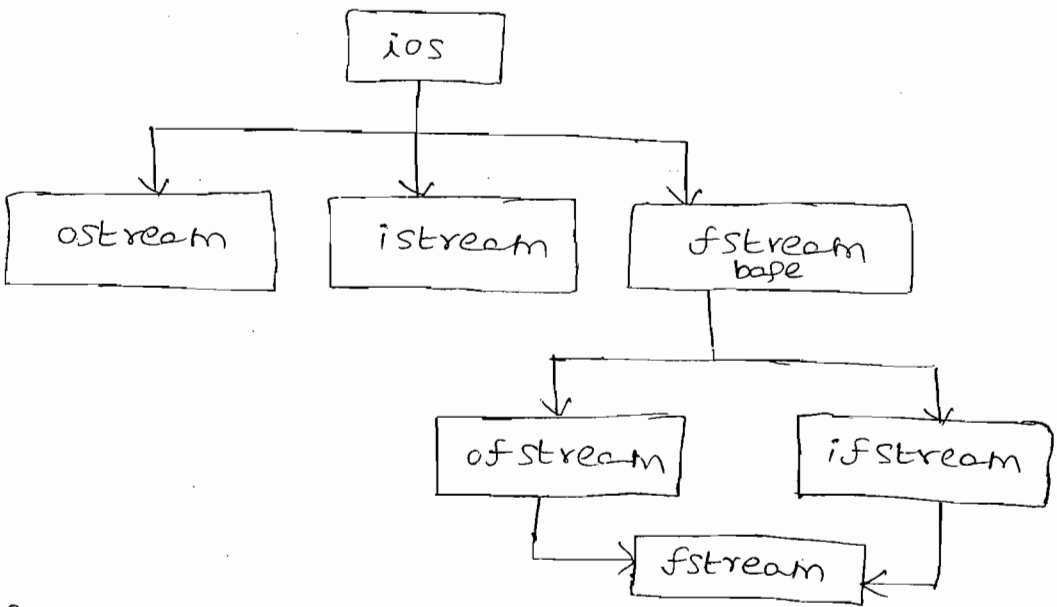


output

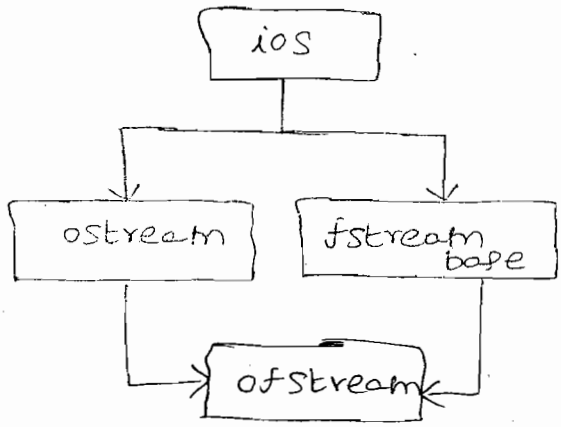
```
Input Any String: Naresh
N
Na
Nar
Nare
Nares
Naresh
```

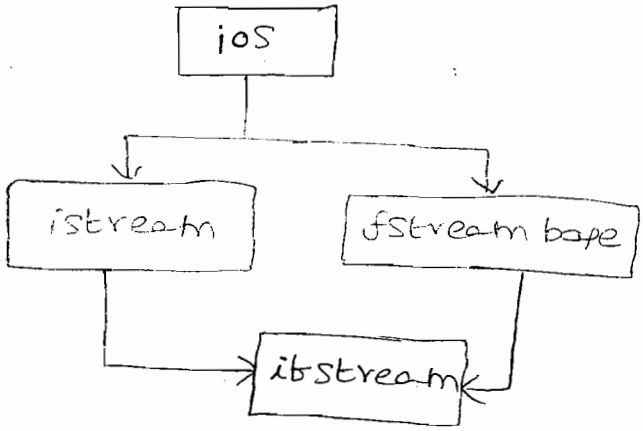
Files :

* 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 ifstream, ofstream, fstream.
* These are declared in fstream.h
* ifstream is nothing but a input file stream.
* ofstream is nothing but a output file stream.



* ofstream is a derived class of fstream base and ostream classes.
* Ifstream is a derived class of fstream base and istream classes.





* ios and fstreambase are called abstract classes

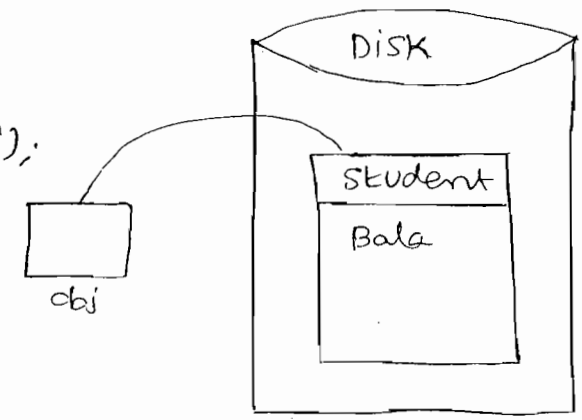
ofstream :

* output file stream class, which provides operations (or) functions to perform output (writeing) operations on file/disk.

* ofstream creates new file for writing data.

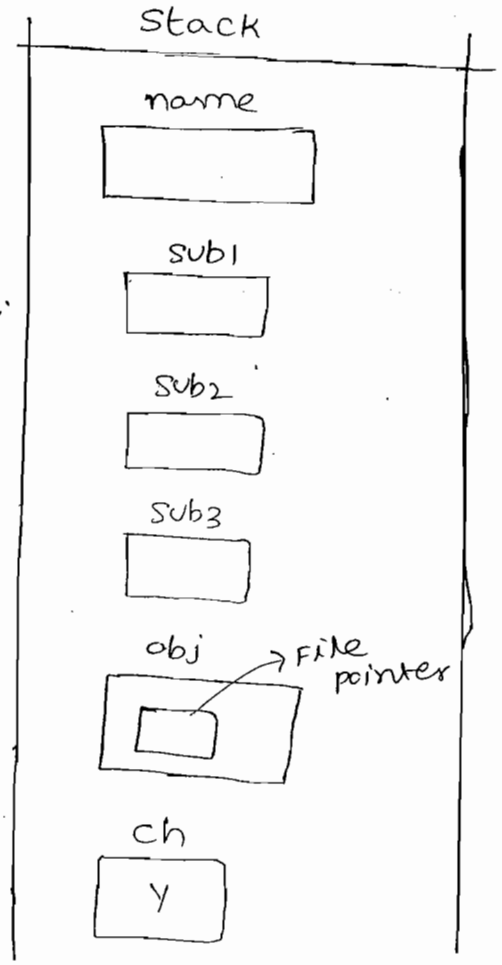
* open is a member function of ofstream class. which open the file in output mode for writing data.

```
ofstream obj;
obj. open ("Student");
obj << "Bala";
```



program :

```
#include <fstream.h>
#include <iostream.h>
#include <conio.h>
int main ( )
{
  char name[20];
  int  sub1, sub2, sub3;
  ofstream obj;
  char ch='y';
 obj.open ("mark1"); clrscr ();
 while (ch != 'n')
{
cout<<"\n Input name:";
cin>> name;
cout<<"\n Input 3 subject mark1:";
cin>> sub1 >> sub2 >> sub3;

 obj <<name<<endl;
obj <<sub1<<endl<<sub2<<endl<<
                sub3<<endl;
cout<<"\n Add Another student ?";
  cin>> ch;
}
  return 0;
}
```




```
[ go to File → DOSshell

c:\TCC>type mark1 ]
```

To see output :

To click on file to select Dos shell

c:\TCC> type mark1

ifstream :

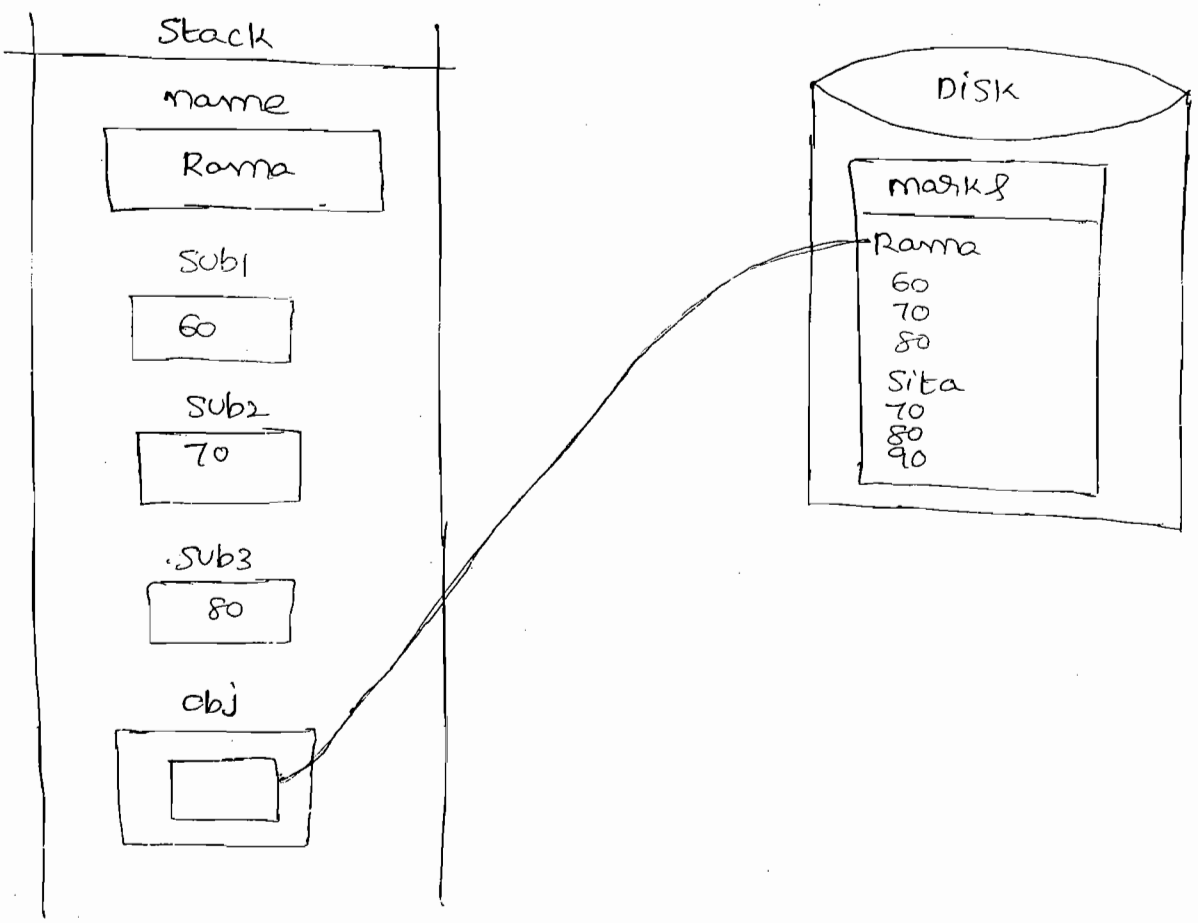
* It is called as input file stream, which provides member functions to read data from existing file.
* ifstream opens the file in input mode (reading).

open() :

which opens the file in input mode.

open ("file name")

If file does not exit. it returns null



Program :

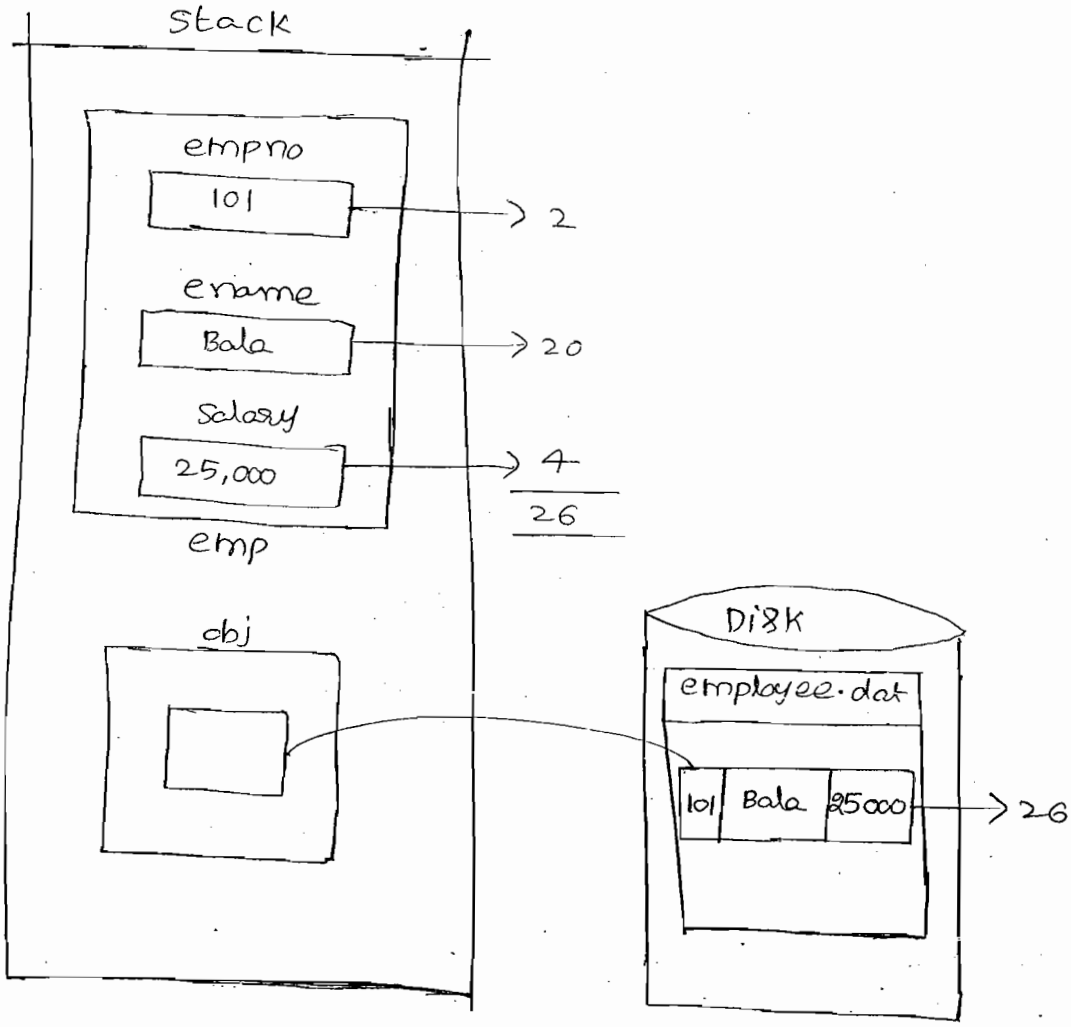
```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
int main()
{
char name[20];
int sub1, sub2, sub3;
ifstream obj;
obj.open("marks");
clrscr();
while(1)
{
obj>>name;
obj>>sub1>>sub2>>sub3;
if(obj.eof())
break;
cout<<"\n"<<name<<" "<<sub1<<" "<<
    sub2:<<" "<<sub3;
if(sub1<40 || sub2<40|| sub3<40)
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<<"\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 ( )
{
 employee emp;
  ofstream obj;
   obj.open("
```

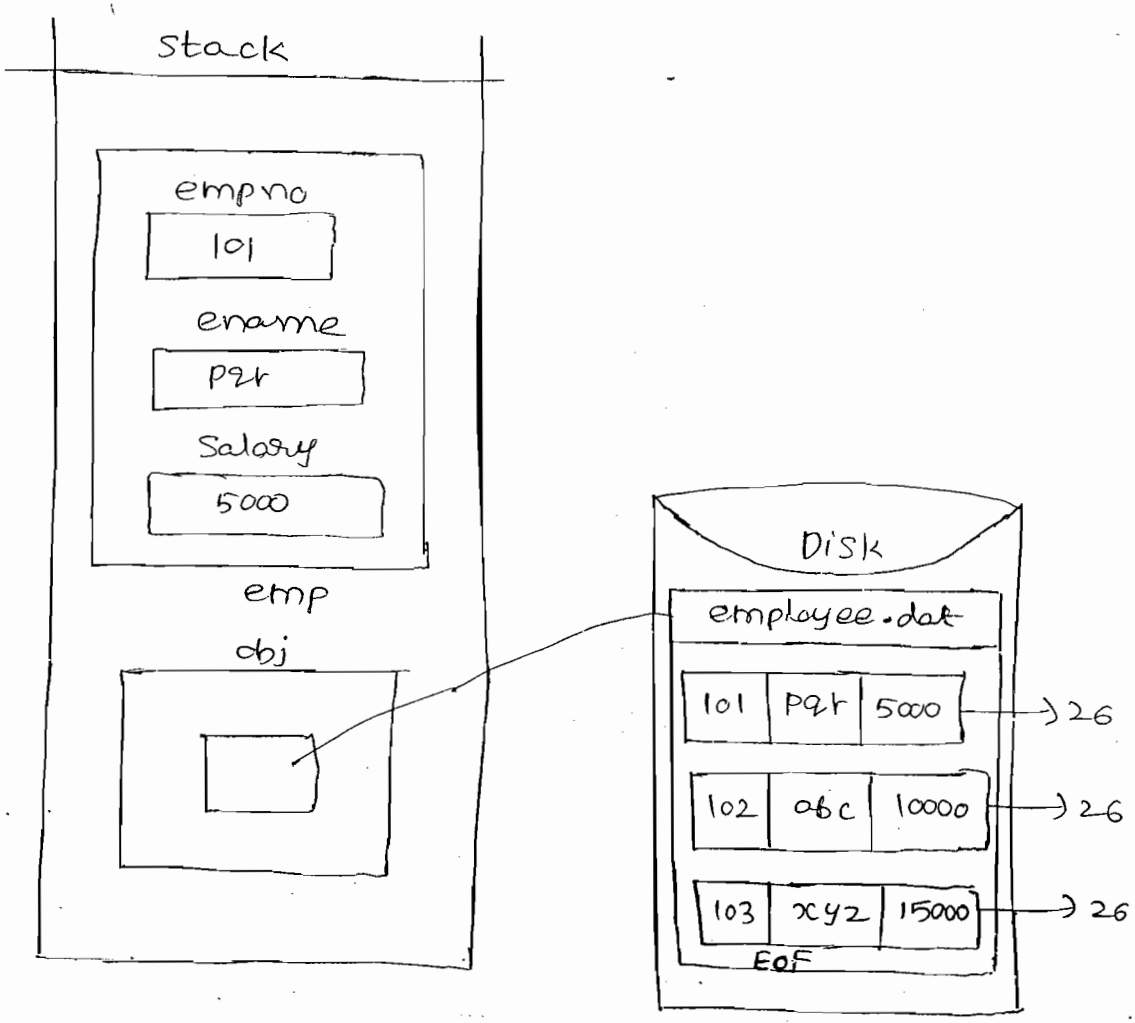
```
char ch='y';
  clrscr();
  while(ch!='n')
  {
     emp.read();
     obj.write((char*)&emp, sizeof(emp));
  cout<<"\n Add another employee?";
   cin>>ch;
  }
  obj.close();
   return 0;
}
```



Reading objects from existing file (or) Records from existing file :

```
#include<fstream.h>
#include<iostream.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()
{
employee emp;
ibstream obj;
clrscr();
```

```
obj.open ("employee.dat");
while (1)
{
  obj.read ((char*) &emp, sizeof (emp));
   if (obj.eof ())
     break;
   emp.print ();
  }
   obj.close ();
   return 0;
}
```



manipulation of file pointers :

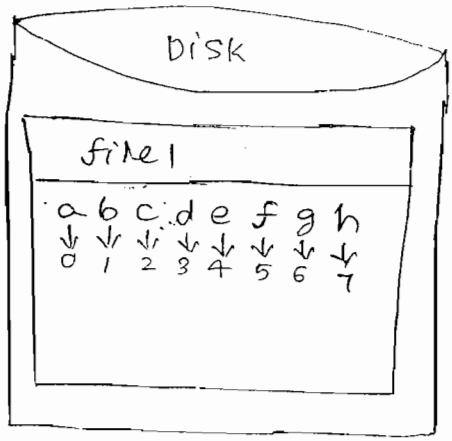
* we can move the file pointer to any desired position in the file. we can take the control of the file pointer.
* The functions to position the file pointer at desired position.
* seekg (offset, position), seekp (offset, position);
  offset parameter represents number of byte and position parameter take 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.seekg (1, ios::cur);
obj.seekg (-2, ios::cur);
obj.seekg (-4, ios::end);
```

obj



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 ofstream and itstream.
* fstream allows to perform input and output file operations.

open ( ) :

* member function which opens the file inparticular mode.

Syntax :

```
open ("filename", file-open-mode);
```

PARAMETER :	MEANING :
Ios :: app	append to end of file
Ios :: ate	goes to end of file.
Ios :: binary	binary file
Ios :: in	open file for reading only.
Ios :: nocreate	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  datamember.
                                  ↑      ↑
obj.open ("Student", ios::out);
obj.open ("Student", ios::app);
obj.open ("Student", ios::out | ios::app);
```

obj.open("student", ios::out | ios::nocreate);

objopen("student", ios::out | ios::in);

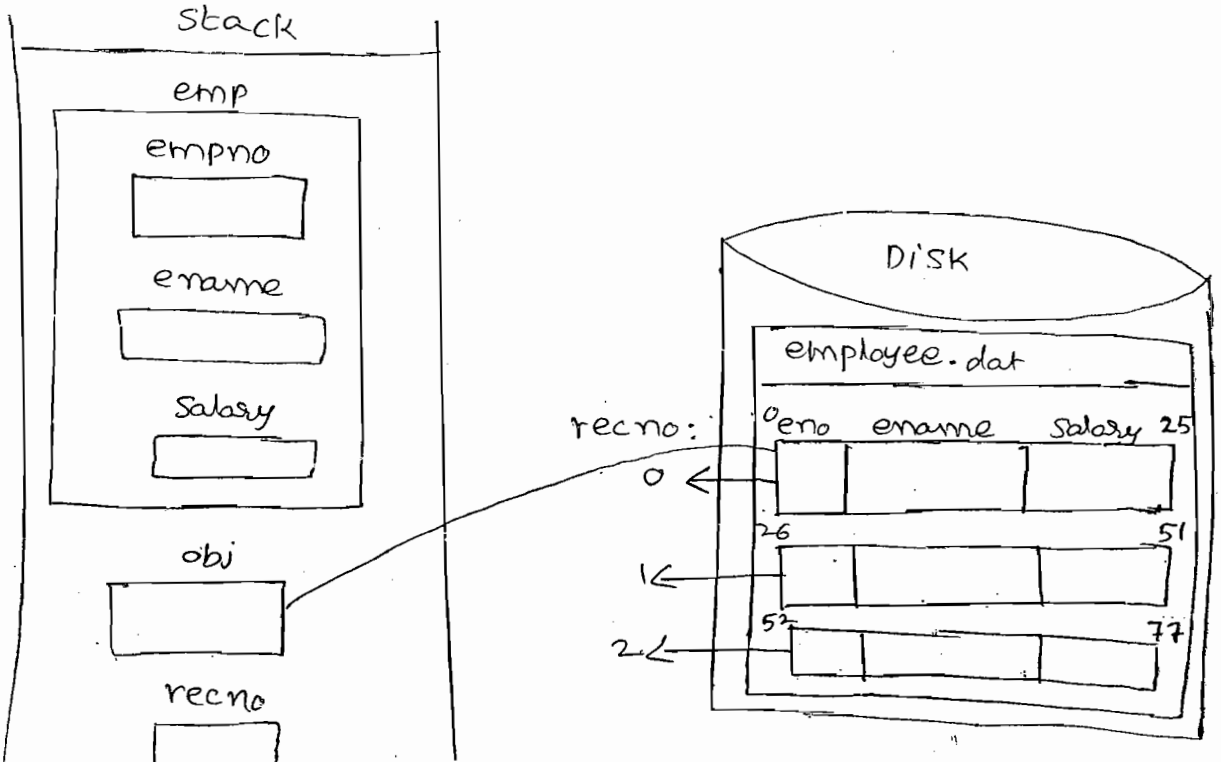
Reading / Writing
Replace / Updation.

Program:

Reading employee objects Randomly.

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class employee
{
 int empno;
 char ename[20];
 float salary;
  public:
  void read()
  {
   cout<<"\n Input employee no:";
   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;
  obj.open("employee.dat", ios::in);
    int recno;
      clrscr();
  cout<<"\n Input record number:";
   cin>>recno;
    obj.seekg(recno*sizeof(emp), ios::beg);
  obj.read((char*)&emp, sizeof(emp));
  if(obj.eof())
  cout<<"\n record not found";
  else
    emp.print[redacted]();
    return 0;
}
```



program :

updating/ replacing object

```
#include <fstream.h>
#include <iostream.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);
  clrscr();
cout<<"\n 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 interrupts are called Asynchronous Exceptions.
* Exception is nothing but a simple logical error.
* Exception is a simple logical errors 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 4.5

* Exception is a Runtime error when it occures 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 occures 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
{
    Bussiness logic;
}
catch (type parameter-name)        → variable-name
{
```

```
catch (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 (type1 arg)
{
   // catch block
}
catch (type2 arg)
{
  // catch block
}
```

Program :

```
#include <iostream.h>
#include <conio.h>
  int main ()
  {
     int a, b, c;
       clrscr ();
       cout<<"\n Input a, b values:";
       cin>>a>>b;
       try
       {
          if(b==0)
```

```
        throw o;
      else
        c = a/b;
        cout<<"\n c= "<<c;
      }
      catch( int x)
      {
       cout<<"\n [redacted] Divide Error \n";
       cout<<"\n "<<x;
       }
        return 0;
}
```

program :

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
 int main()
{
  char uname[10], password[10];
     clrscr();
   cout<<"\n Input username :";
    cin>> uname;
   cout<<"\n Input password :";
    cin>> password;
    try
    {
     if(strcmp(uname,"nit")==0 &&
          strcmp(password,"nit")==0)
      cout<<"\n welcome to my application";
      else
```

```
    Throw 'f';
  }
  catch (char x)
  {
cout<<"\n Invalid username or password";
  }
   return 0;
}
```

Program:

```
#include <iostream.h>
#include <conio.h>
 int main ()
{
  int x;
   clrscr ();
 cout<<"\n 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<<"\n value is zero";
 }
 catch (float b)
 {
  cout<<"\n value is -ve";
 }
 return a;
}
```

---

name spaces:

* namespace is a collection of variables and datatypes (class or structures).

* name space is a container.

Syntax:

```
namespace <name space - name>;
```

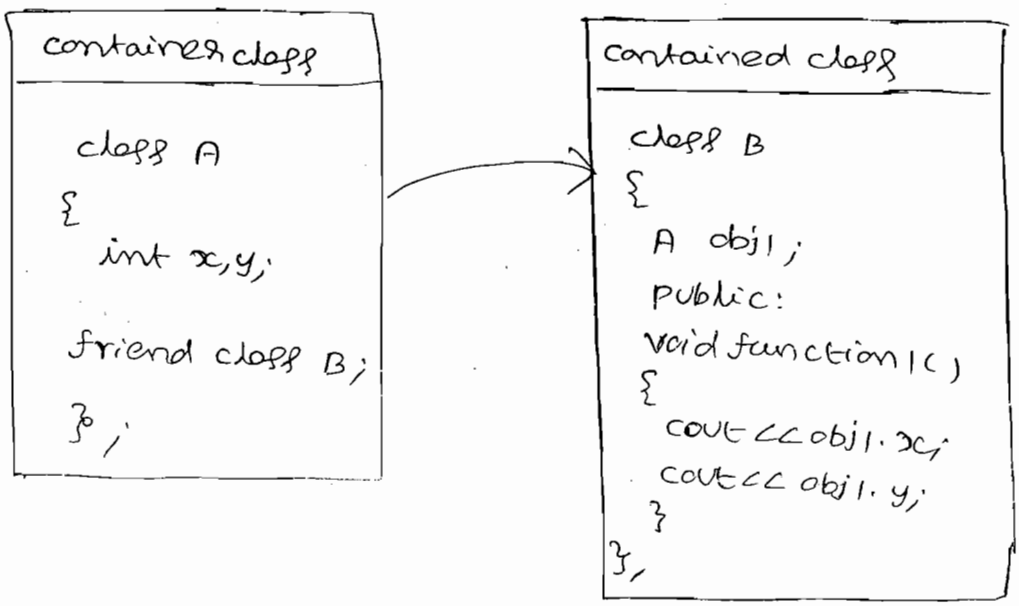
* namespaces does not support on TURBO 3.0 and 4.5 support for DEV C++

Ex:

```
namespace A
 int x;
 int main()
 {
  cout<<A::x;
 }
```

```
using A;
int main()
{
 cout<< x;
}
```

Container 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 *argv[])
{
```

```
clrscr();
cout<<"\n Argument count "<<argc;
cout<<"\n"<<argv[1];
cout<<"\n"<<argv[2];
return 0;
}
```

[To save ex1]

```
C:\TCC> ex1  10  20
```

output:-

go to dos shell

```
C:\TCC> ex1  10  20

Argument count 3

10
20
```

Graphics

* Graphics does not support Turbo 4.5 It support only Turbo 3.0

initgraph() :

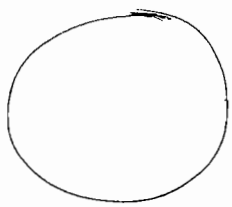
* init graphical screen.

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;
}
```