C++

BY.KIRAN

NARESH TECH

FRIENDS XEROX

Gayatri Nagar,Ameerpet.Hyd

Cell:9985011311

	page no.
1. Introduction of C++.	1
R → i/p, o/p Statements.	4
→ Declaration of variables.	5
→ C++ Operators.	8
2. Arrays.	9
3. pointers.	11
4. Functions	14
→ Differences b/w Normal function & Inline function.	25
→ Diff b/w Macros & Inline.	
5. function overloading	25
→ Diff b/w C & C++	29
6. OOP's. (Object Oriented programming)	31
→ Encapsulation.	
→ Data abstraction.	
→ Class.	
→ Object	
→ Private, public.	37
→ "this" pointer.	
7 Constructor & Destructor.	45
→ Static Datamembers & memberfunctions.	
8. Inheritence	61
→ Granting Access.	70
9. Friend function & Classes.	73
→ Advantages of Inheritence.	81
→ Container Class. (OR) Compositional Class.	

10. polymorphism 84
→ Operator overloading.
11. Type Conversions. 92
→ Basic to Class
→ Class to Basic
→ Class to Class.
12. Runtime polymorphism. 95
→ Virtual Function.
→ Abstract Class.
→ Constructors & Destructors in Late binding. 101
13. Multiple Inheritence 104
14. Exception Handling. 105
15. Templates. 109
16. Streams & Files. 111

Drawback: oop's don't Not Support to develop Embeded application by using 'c' we can develope it..

oops- developer = Grady Booch.

C++ - is an object-oriented programming language. where objected oriented is a Concept (or) method (or) principle (rules & legulations)

(C++) Always concept follows rules & legulations

(C) Language follows Syntax.

A problem has one solution but problems has diff approches. They are.

1. Structure oriented programming, (procedure oriented) also.
2. Assembly programming.
3. Machine oriented "
4. Objected oriented programming.

1. What is a program?

a program is set of data & instructions.

(or)

a program is a collection of stmt's & procedures effecting on data(variables).

Programming: programming is an organisation of Data & Instructions according to the given problem.

Programming Concept: This are set of Rules & Regulations to Organise data & Instructions.

Programming principles was developed to Reduce ~~peiko~~ Space & time Complexity.

Programming Language:

a programming Language is a tool for the development of software. which provides set of predefined instructions of the programming Language consist of words, digits & simbols.

~~Ist C~~

1. Is ft C & C++ Compailer is Same (or) Not?

A No.

— The 'C' Compiler is diff & C++ compiler is diff.
a 'c' program will be executed by 'c' compiler.
a 'C++' program will be executed by C++ compiler.

Ex: prof: Main()
{
int a;
Char ch;
float class;
}
Variable in 'c'

.c
Execute.

Main()
{ int a;
char ch;
float class;
}

.cpp
Error
Not execute.
↳ Here in C++ class is predifined keyword.

Note: — The c++ .Exe file is More optimised (fast) than execution.
— the 'c' o/p file

→ Any were works.

→ For the Reason of portability we have Two Compiler's available 'c' is separate & c++ is separate.

Programming paradigms:
↳ (Standards (or) patterns)

The programming methods (or) Standards defines diff way of style of programming depends on the patterns of the programming we have diff Catagories.

1. Monolithical programming
2. procedural
3. Modular programming
4. Object oriented programming.

1. Monolithical programming:

In Monolithical programming the programming stuff's will be executed only one time.

If is a Sequence of programming every stuf- More than one time can't execute.

Drawback:

Before procedural programming when we implementing a program the major drawback we find Redandency.

Repitation of Same Stmt's more than one time is Redandency.

if One operation performed more than one time there is a chance of Redandency. which altimatly increase the size of the program. If is an inefficient programming technic

scanf( )
f = fainal specification.

2. Procedural programming:

a procedural programming is organisation of data & instructions by dividing into small module, implementing the programming called as subroutins, (or) procedures (or) functions.

Characforsics of procedural programming:

1. Large programs are divided into Small programs.

Struct- Railway-ticket-
{
Char name[20];
Char * date;
Unsigned age;
char gender;
}

Age only +ve value.

→ The fact program is an application.
Sum of n's is an appliation.
project is a Collection of application

2. Most of the Functions share Global Data

3 - Data moves openly around the system.

(1) SW modules program means every ex: turbo SW program save with .h. & use these files in main program with calling of SW programs through file.h calling like #include "file.h"

```
struct Railway_ticket
{
  char name[20]; // Global Data.
  ===
  ===
}
main()
{
  ===  // functions.
}
```

4. Functions transform data from one to another. (through arguments)

5. Importance is given on functions rather than Data.
   → Second class treatement is given for Data.
   → because Small Change in Data, it will effect the all functions.

```
ex: int a;
    float b; } change → effect.
    scanf("%d %f", &a, &b);
```

In the above change the a Data, i.e 'a' data type, it's effect's on scanf() function specifications.

Texas A&M University

AT&T Bell lab's → American Telephonic & Telegraphic Bell lab's.

Advantages: 1. Modularity.

Dividing the program instructions according to it's operations is called as Modularity. (This Modularity is Done through functions.) → SW programs.

2. Reusability:

Write the code Once & use if more than One time, is called as Reusability.

∴ Reusability avoid Redundancy → duplicates.

3. Simplicity: Easy to understand & debug → Checking errors.

4 Efficiency: It is an efficient programming Technic.

EX: Cobol & Bottom. Fortran

'C' is an procedural, modular programming & Structure oriented programming.

Disadvantages:

History of C++

~~Steams~~:

Input, output Stuff's:

Every programming language will have i/p, o/p statements inorder to capture the data from the i/p devices & inorder to display the data to the o/p devices we have o/p stmt's.

Input: Data (or) information is given to the program is called as input. (Data which flow inside the program is called as input).

Output: Data (or) information given by the program is called as output. (Data which flow outside the program is called output).

All i/p, o/p operations must be handle through External library files in c++. (Stream Library)
↳ collection of things.

Stream is general flow of data.

<stdio.h> → contains declaration of functions in 'c'

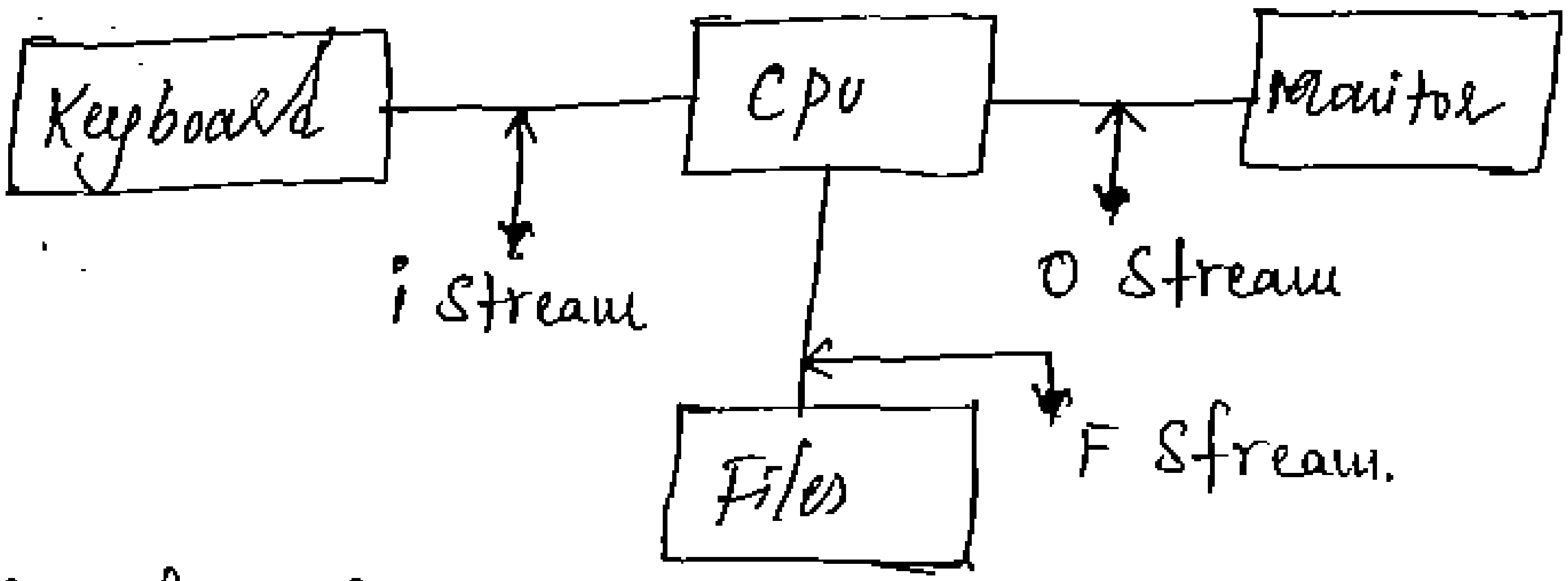
→ a Stream contains of diff. kinds of data flow's. Each Stream is associated with a class.

→ a Stream is a Sequence of bytes if acting as a source from which the input data can be obtained (or) a destination to which the output data can be send.

→ The Source Stream that provides data to the program is called as Input Stream. (i Stream)

→ The Destination Stream that receives the o/p from the program is called as Output Stream. (o Stream)

→ A Stream act's as an Interface b/w the program & the i/p o/p devices

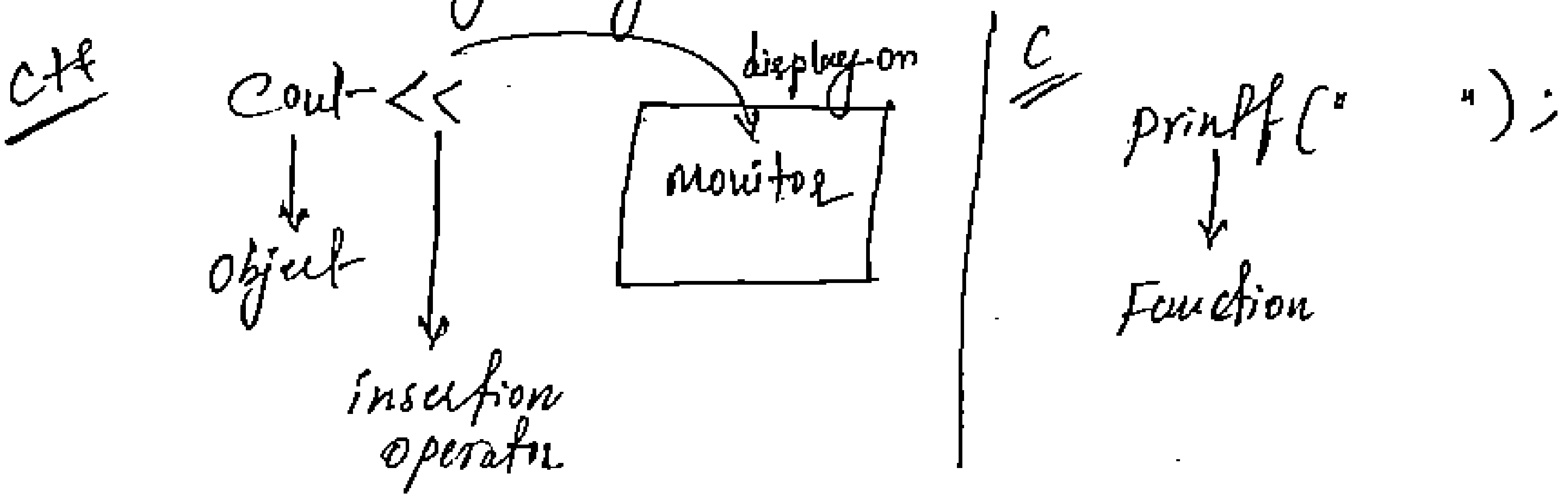


Output Statement:-

In order to display the output in C++ we use one output object with an operator **cout**.

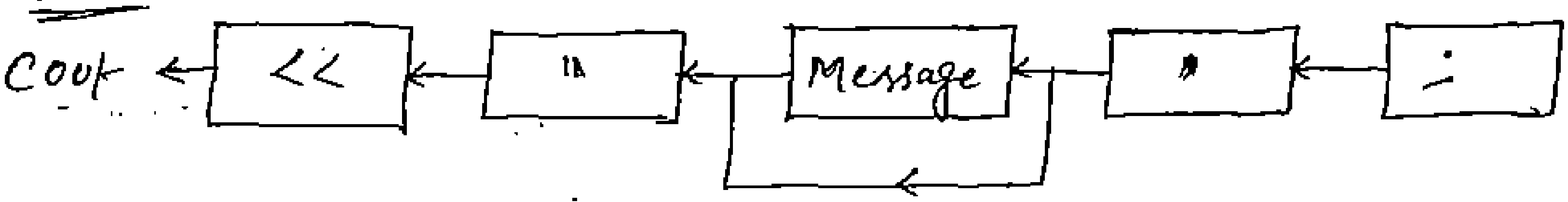
→ str screen.
C = Console (or) Cascading.

cout is used to take any information onto the o/p stream by using an operator insertion operator. <<



Syntax Diagram:

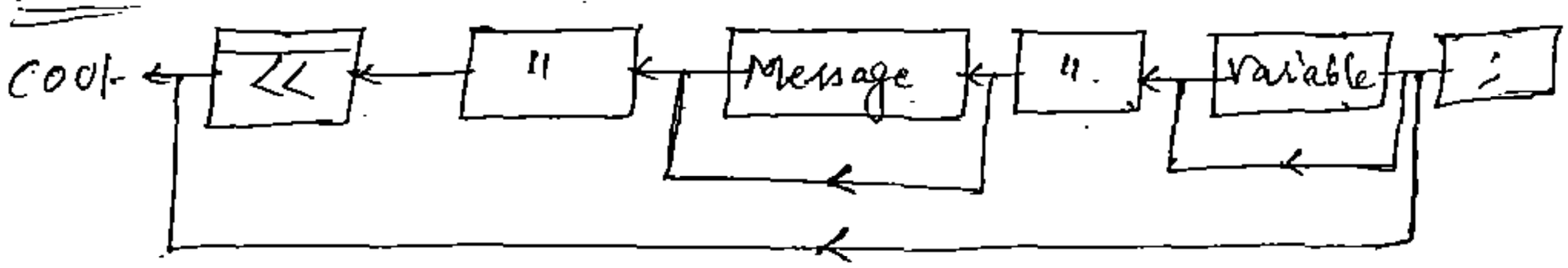
Form 1:




```
#include <iostream.h>
#include <conio.h>
void main()
{
clrscr();
cout<<"Welcome to C++";
getch();
}
```

O/P: Welcome to C++.

Form2:



C

```
int-
float a=10;
float b=67.86;
P.f("A:%d S:%f", a, b);
```

→ When a type change to float, o/p will effect.

```
int- a=10;
float- b=67.86;
Cout-<<"A:"<<a<<"S:"<<"S
Cout-<<"A:"<<a<<"S:"<<s;
```

→ C++ Compiler internally convert it.

→ The Stream concept of i/p o/p don't require any format specifications to perform i/p o/p operations.

→ The Conversion is an implicit function automatically takesplace in C++.

→ The cout is an implicit object whose declaration is available in the iostream.

→ cout performs o/p operations both formated and unformated

→ Cout represent's Standard o/p device called as Monitor.

→ cout doesn't require any format Specifiers for performing o/p operations.

→ Cout uses an operator called as insertion to print the data on Monitor.

→ Cout can display Variable type of o/p's, Constant no's, & expressions.

```
Cout-<< a; // Variable
Cout-<< 5; // 5 is Constant
Cout-<< 5+2<<a+b; // expressions.
```

Cin :-

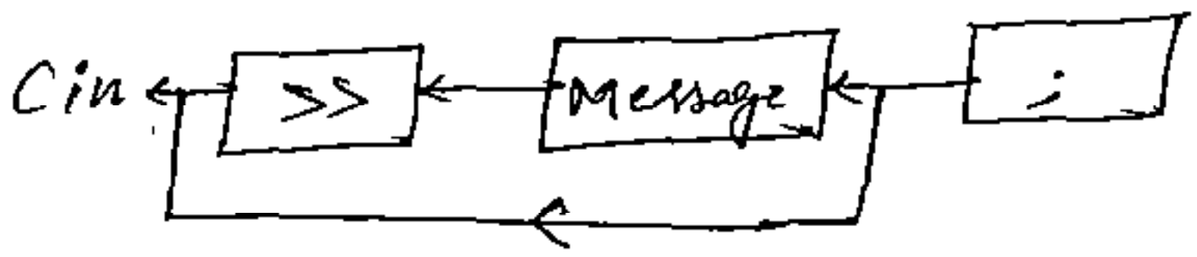
Scanf("formate specification", &arg1, &arg2, .. &argn);

- 'c' lang

' C++ '

→ The cin is used to accept accep information into the memory location ~~to the~~ through the standard i/p device.

Syntax Diagram:



→ At the time of accepting our i/p stmt by ~~Scanf~~ Cin we don't need to specify any formate specifications. & address symbols.

→ The cin stmt performs ~~for~~ formated & Unformated i/p operations. ~~Cin~~

→ Cin Represents the standard i/p device called keyboard.

→ Cin read the data from the keyboard, during execution of the program.

Declaration of a Variables:-

In 'C' lang declaration of variables can be takes place only in declaration block. In a Program whenever the scope is starting the 1st block is called as declaration block. & then Execution block will takesplace.

— There are diff types of Variable declaration blocks

Variable
1. Conditional block
2. Iterative block.
3. Function block 4. Anonymous block

```
Main()
{
 Int a;  } Declaration block.
 float b; }
 Classes();
 printf("____");
 scanf("____");
 int c; // Error in 'c' lang.  // In C++ it is valid
 ==                              anywhere we declare.
}
```

Variable declaration only declare in declaration block only.

After declaration Execution statements are 'c' lang.

```
Main()
{
 int a;
 Classes();
 if (condition)
  { int a; // valid in 'c' & 'C++'.
  ==
 }
}
```

```
Main()
{
 int a;
 {
  int a;
  {
   int a;
  }
 }
}
```

Anonymous block

Block has No name anonymous block

```
Main()
{
 int j;
 Classes();
 for(int i=0; i<=5; i++)
 {
 }
}
```

Error in 'c' lang → In C++ it doesn't contain any specific declaration block

→ 'C' header files contains declarations

→ C++ header files contains Partial definitions.

1, 2, A, ... = Data keys

pageup, dparrow, f_1, f_2 ... = Non Data keys

→ getch() will load the datakeys & Non Datakeys.

→ getches used to wait in dos program or wait until press any key otherwise not wait.

Manipulators: How the 'c' being supports Escape Sequences for designing the o/p's c++ also supports formated Console i/o Operations for designing the o/p in 3 Ways.

1. io's class functions & flags.
2. Manipulator's.
3. User named output functions. (User define their own Escape sequences)

Manipulator's:- Manipulator's are the instructions to the o/p Stream to modify the o/p in diff ways.

The manipulator's provide a clean & easy way for the o/p. When we using manipulator's the formated instructions are inserted directly into the stream.

Manipulator's of 2 types.

One can accept argument's.

Some Manipulator's doesn't accepts arguments.

Manipulator's will be define under <iomanip.h>

The Manipulator's are more flexible, at a time we can use any no of Manipulator's in a Single line.

endl: → ('\n')

inserts new line and flush Streams.

endl will not occupy Memory Space. [Same job for \n & endl]

\n — will occupy one byte

setw(int w) : — (\t) w = width.

sets the feald ~~with~~ width to w.

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

void main()
{
int a=10000; float b=56.676;
clrscr();
cout<<" Welcome to C++"<<endl;
cout<<a<<"\t"<<b<<endl;
cout<<a<<setw(25)<<b;
float s=789.4646895485;
cout<<s<<setprecision(3)<<s<<endl;
cout<<a<<setw(20)<<setfill('$')<<b
                 <<"\n";
cout<<"Octal "<<setbase(8)<<b;
cout<<" Hex "<<setbase(16)<<b;
getch();
}
```

setprecision will display upto 2 decimalpoints. in 'C' using p.f(%.2f);

setfill :- set the filling of characters.
always place the arg's in single quates.''

→ setfill() has to must Use with setw() fill the gap

setbase() :- This Manipulator used to display the number at in octal, hexadecimal formates.
in 'C' use %o - to octal
%x - Hexadecimal
%#o
↳ display prefix.

C++ Tokens:

A Smallest unit of a program is called as Token.

The C++ tokens

1. Identifiers
2. Keywords (64) (32 old + 32 New) added according to ANSI
3. Constants.
4. Variables.
5. Datatypes.
6. Operators. (44 in 'C' lang. + 3 added in C++)
7. Special Symbols.

Identifiers: An Identifier is a Name given for Variables, Constants, Macro's, *. etc.
array names,
If is a user defined name.

Variable:
a Name which is given for any Computer Memory location is called as Variable.
The purpose of the Variable to Store Some data.

Datatypes:- a Datatype describes what type of data, we can Store in a Variable. if also describes the size of the Variable.

Syntax:

<type> <size> <Sign> datatype Val_name = <initialisation>;
↓ like alter the meaning
↳ Short, long.
+ −

type, Size, Sign are optional.

Keywords:- a Keyword is a reserved word, in which meaning already defined by the compiler. As per the C++ language there are 64 keywords.

Ex:- virtual, this, class, protected, public, new, delete, ... etc.

Constants:- a Constant is a Value that the doesn't change during the execution of the program.

Operators:- An operator is a Symbol which will be manipulated On Operands.

Precedence press Ctrl + F5
↳ to find the No. of operators in C++.

C++

C++ operators :-

:: Scope Resolution operator

new } dynamic memory
delete } Allocation / deallocation.

C++ Deference operators:

.*

->*

:: (Scope Resolution):

local variable & global variable are present with the same name always the priority will takes place only for local variable

```
#include <iostream.h>
#include<conio.h>
int a=10; // Global variable
int main()
{
 int a=67; // local variable
 cout<<a<<endl;   67
 getch(); return 0;
}
```

basically local variable value, will be the Garbage due to the Storage class Specifier is auto

In 'c' lang Whenever user want's to display both the local variable & the global variable at a time, if is not possible, in Single function.

But in c++ to give the flexibility of accessing global variable & local variable is possible by the :: scope Resolution Operator

Syntax: :: Var name;

```
#include <iostream.h>
#include <conio.h>
 int i=10;
 void main()
{  int i=5; clrscr();
   int i=5;
cout<<" localvariable : "<<i<<endl; 5
cout<<" Global variable : "<<::i<<endl; 10
cout<<"sum :"<<i+::i<<endl;
                          15
getch();
}
```

```
int i = 20;
main()
{
  int i=5;
cout<<"i<<endl; 5
cout<<::i<<endl; 20
{
  int i=25;
  cout<<i<<endl; 25
  cout<<::i<<endl; 20
}
cout<<i<<endl; 5
cout<<::i<<endl; 20
}
```

→ The :: (Scope Resolution) operator will also provide of accessing the class member functions by defining outside a class with this operator.

We can access the member-functions outside the class with help of classname as well as :: (scope resolution) operator.

→
```
#include <iostream.h>
#include<conio.h>
int a = 5;
char str[10] = "hai";
void main()
{
  clrscr();
  char str[10]="hello";
  int a=6;
  { //block1
      char str[10] = "everybody";
      int b=a;
      a=7;
      cout<<"\n innerblock-2";
      cout<<"\n b:="<<b; 6
      cout<<"\n a:="<<a; 7
      cout<<"\n::a:="<<::a; 5
      cout<<"\n\nstrings";
      cout<<"\n----";
      cout<<"\n Content of globalstring:"<<::str; hai
{char str[10]="Inn";
cout<<"\n In inn.."<<str; Inn
cout<<"\n enter any value for global variable";
cin>>::a;        hai
cout<<"\n Value of global variable"<<::a; 5
}
}
cout<<"\n\n in inner-1 block";
cout<<"\n a a:="<<a; 7
cout<<"\n str:="<<str; hello
}
```

→ Here a is not declared because declared by in outside already in outside value use inside the block

Arrays:

Def: An array is an single subscripted variable, which can hold 'n' no. of elements.

syntax:

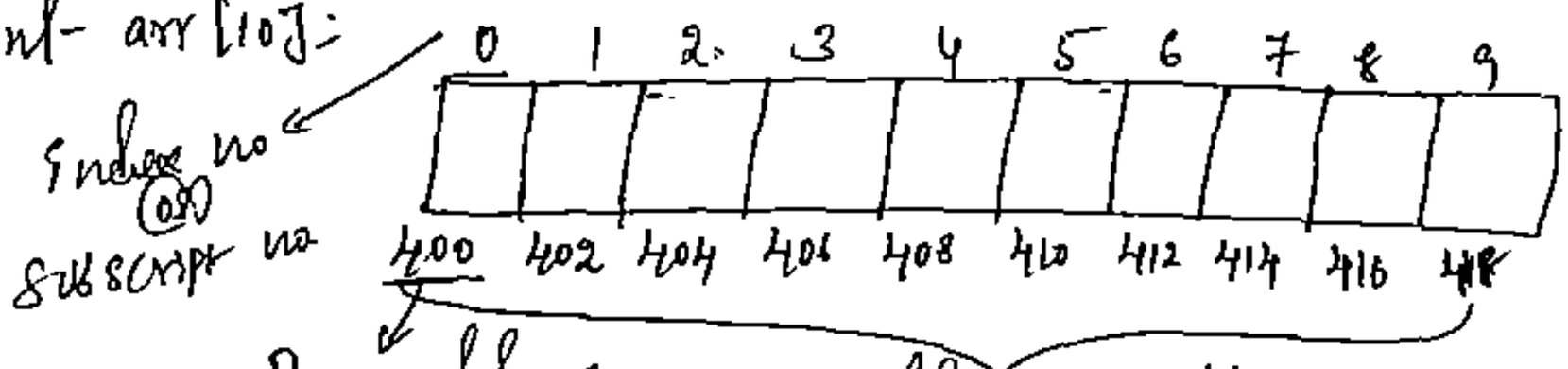
data-type Var-name [size]; → no. of elements.

Dimension.

Whenever an array variable is Declaring the size of the variable must be define, without defining the size we can't declare an array variable.

An array is an homogenious type, which can accept same type of elements.

→ int- arr [10];



Whenever an array variable is defining the size must should be Constant-value.

An array index always starts from '0'.. a Normal user will access the array by the subscript no.

→ a[10]
*(a+10+10)

- [internally there is No Subscript-No.'s & Only Address
The Compiler will access by the addresses. The addresses of each block will be sequentially located to access the memory blocks.

Arrays are Classified in Three Dimensions.

1. Single Dimention array
2. Double Dimention (2-array)
3. Multi Dimention.

Single Dimentional Array

int a[5];
float s[20];
char name[25];
} declaration.

int a[5] = {10, 20, -5, 4, 9}
Initialisation at the time of declaration.

float s[8] = {1.2, 3.5, 6.9, 9.6}
Here 4 blocks are accessed remaining blocks are wastage

(or) char name[25] = {"Kiran"}
char name[25] = {'K', 'i', 'r', 'a', 'n'};

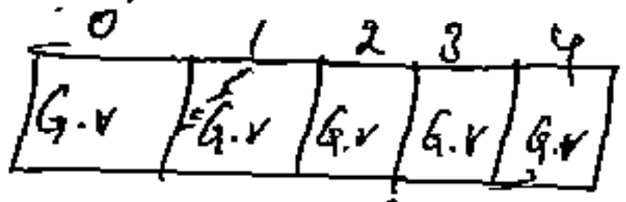
int arr[ ]; // invalid
Without defining the size off the array there is an error
int arr[n]; // invalid 'n' is Constant Value.

int a=10;
int arr[a]; // invalid because array size always fixed, once we decl but here 'd Value is Changed because if is a Variable.

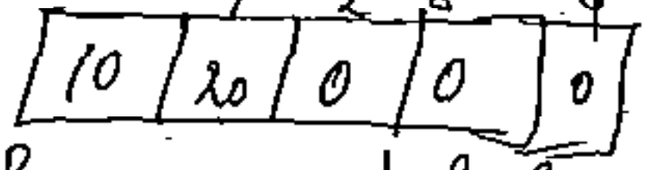
const int a=10;
int arr[a]; ✓ possible

int arr[0]; // invalid
Array Size is Not Zero.

int arr[-5]; ✗
int a[2000]; ✓
long a[2000]; As per 16 bit Not possible, As per 32 possible
int a[5];



int a[5]; {10,20}



At the time of declaration if you initialise an array variable with atleast one element the remaining elements are Zero's

#define M 10
int arr[M];

⟹ Write a prog accept elements in an array & display them.

```
#include <iostream.h>
#include<conio.h>
void main()
{
 int a[10]={0}, n;
 clrscr():
cout<<"Enter the value of n:";
cin>> n;                          // Here 'i' indicates = index no.
 for(int i=0; i<n; i++)                  a[i] indicates = Element.
  {
    cout<<"Enter the value at a[" <<i <<"]" <<":";
    cin>> a[i];
  }
 for(i=0; i<n; i++)
  {
   cout<<"a[i] <<" ";
   getch();
}
```

⟶ Write a prog accepts some elements in array, delete an element from an array.

0	1	2	3	4	5	6	7	8	9
22	45	67	45	39	G.V	G.V	G.V	G.V	G.V

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int arr[20], n, del, ck=1,
  clrscr();                    ⟶ n = no. of array elements taken
 cout<<"Enter the value of n:"; 5
cin>>n;
 for(int i=0; i<n;
 {
  cout<<"Enter the value of array at arr[" <<i<<"]" <<":";
  cin>>a[i];
 }
 cout<<"Enter the element to Delete:"; 45
cin>> del;
```

```
cout<<" Before Deleting the element:";
cin>>a[i];
for(i=0; i<n; i++)
  cout<<arr[i];
/*logic*/
for(i=0; i<n; i++)
{   if (arr[i]==del)
    {   ex=1;
      for(j=i; j<n; j++)
        arr[j] = arr[j+1];
    }
    n--;
    i--;
  }
}  if(ex==0)
  cout<<" Element not found:";
  else
  for(i=0; i<n; j++)
  cout<<arr[i];
}
```

0	1	2	3	4	5	6	7	8	9
22	45 67	67 45	45 39	39 G.V.	G.V	G.V	G.V	G.V	G.V

67 39 G.V.

char name[20][50] (double dimentional array)
-> 20 names, each name size is 50 Characters

→ write a prg accept some elements in an array & also accept a position, insert an element in that position.

Pointers

& — address of
* — value at address.
(indirection operator)
(dereference ")
(Object at that location)

} both of them are come to Unary category.

→ Address will be created (or) allocated by Compiler

offset = Address

Segment size = 64 KB.
RAM size is divided in to 16 segments — starts from 0 to 15.

Generally Data stored in Data segment — data address like this.

~~0X8fc1fffff4~~

0X 8fc1 fff4

- 0X — prefix (X = Hexa decimal)
- 8fc1 — Data Segment
- fff4 — offset

```
#include<iostream.h>
#include<conio.h>
Void main()
{ clrscr();
  int a=10;
  cout<<" value of a:"<<a<<endl;          10
  cout<<" Address of a:"<<&a<<endl;       0X8fc1fff4
  cout<<" Address of a:"<<(unsigned) &a<<endl;   65524
  cout<<" value of a:"<<*&a;              10
  getch();
}
```

→ The value at address Symbol always play the role with the address only. (Directly or indirectly)

Ex) cout<<*a; // Invalid

Def pointer: A pointer is a variable which can hold the Address of other variables. it can be ordinary variable, array variable another pointer variable, (or) it can be user defined datatype variable

If can also hold the address of a function.

→ pointer is an derived data type.

declaration:

data-type * var-name=<initialisation>

int *a; // 'a' is a variable of type integer pointer

float *k; // k is a variable of type float pointer.

Char *ptr; // ptr is a variable " Char pointer.

Primary data types	Desired data types
int	int *
float	float *
Char	Char *
⋮	⋮

int i; // 'i' is a variable of type integer

int *ptr; 1. ptr is a variable of type integer pointer (int *)

2. * ptr gives integer

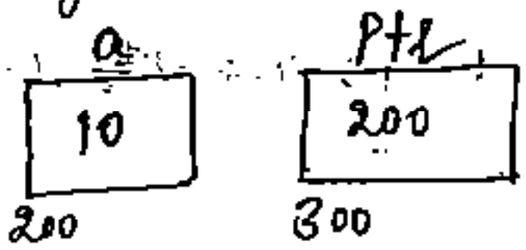
3. int-ptr we can store address of integer variable.

4. ptr is called pointer variable of type integer (or) integer pointer, (or) pointer to integer.

5. This Operator (*) gives the value at the address pointed by the pointer.

```
#include<iostream>
Void Main()
{
    int a=10;
    int *ptr;
    ptr=&a;
    Cout<<a<<"\n"; //10
    Cout<<&a<<"\n"; //200
```

Continuation

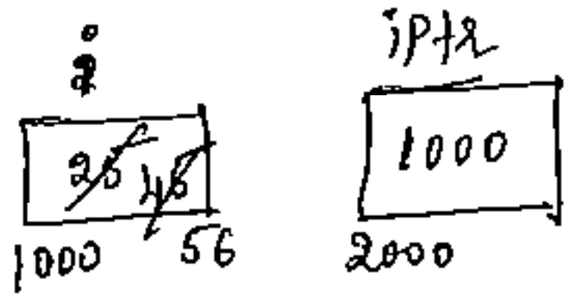


```
cout << *a; // invalid
cout << *(&)a << endl; 10
cout << ptr << endl; 200
cout << &ptr << endl; 300
cout << *ptr << endl; 10
cout << *&ptr << endl; 200.
```

→ *ptr is replace with the object 'a' at that location.

#E iostream.h
conio.h

```
Main()
{
int i = 25;
int *iptr;
iptr = &i;
cout << i << *iptr ; 25, 25
i = 45;
cout << i << *iptr ; 45, 45
*iPtr = 56;
cout << i << *Ptr ; 56, 56
}
```



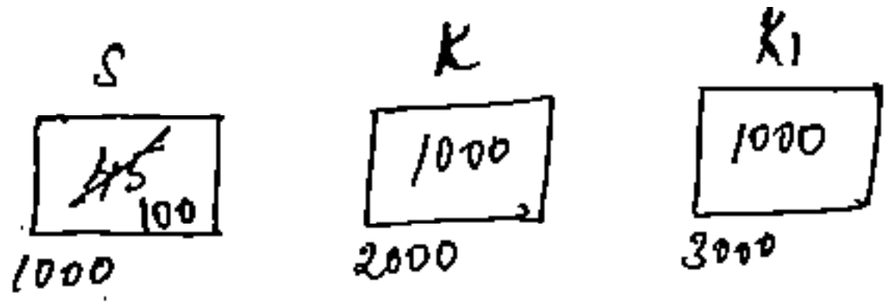
→ In this *iptr is an alias to 'i'. alias to the object of the specific location.
(* operator returns object)

location (or) address

→ All the 43 operators in 'C' language returns a value. only * returns the object: (Address)

→

```
Main()
{
int s = 45;
int *K;
int *K1;
K = &s;
K1 = &s;
cout << s << K << *K1; 45, 1000, 45.
*K = 100;
cout << &s << *K << *K1; 100, 100, 100.
}
```



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

void main()
{
  int *s;
  int a=10;
  s = &a;
  // s=45; //error Can't convert int to int *
  int *s1;
  s1 = &s; //error Can't convert int** to int*
```

Level's of pointer's :-

```
int *p;
int **ptr;
```

**Notes: The size of a pointer variable w.r.t to operating system, irrespective of the datatype.

sizeof( ) : sizeof ~~operator~~ Pointer will returns the estimated ~~value~~ size of datatype w.r.t (or) based on O.S.

```
#include<iostream.h>
#include<conio.h>
void Main()
{ clrscr();
  float *s;
  cout<<sizeof(s)<<sizeof(*s); 2,4
  getch();
}
```

All ~~int~~ datatypes of pointer variable takes 2 bytes in 16 bit O.S. (Turbo C)

Void: ~~Void~~

→ Void is a datatype which doesn't have any size

→ We can't define any variable's with Void.

We can define pointer type of variables by defining void.

Since pointer size depends on w.r.t operating system.

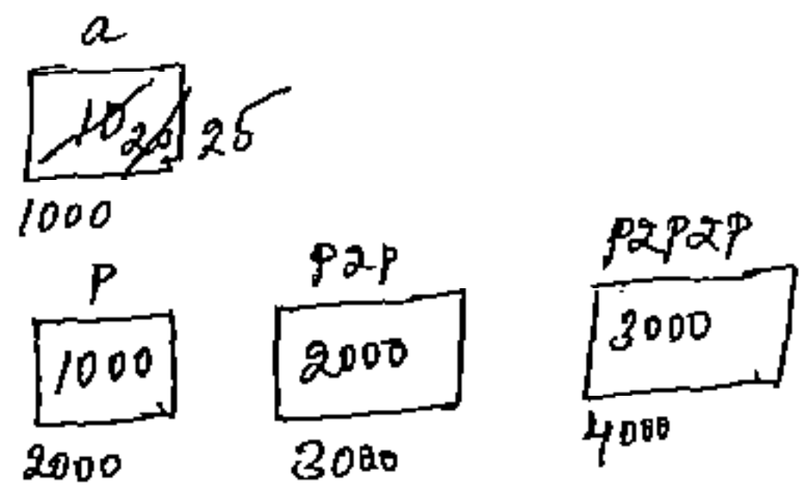
```
#include<iostream.h>
#include<conio.h>
int main()
{ void clrscr();
  void *s;
  cout<<sizeof(s); 2 byte / depends on the O.S. only for pointer variables.
  getch();
}
```

Level's of pointers:

There is no restrictions on the level's of pointers.

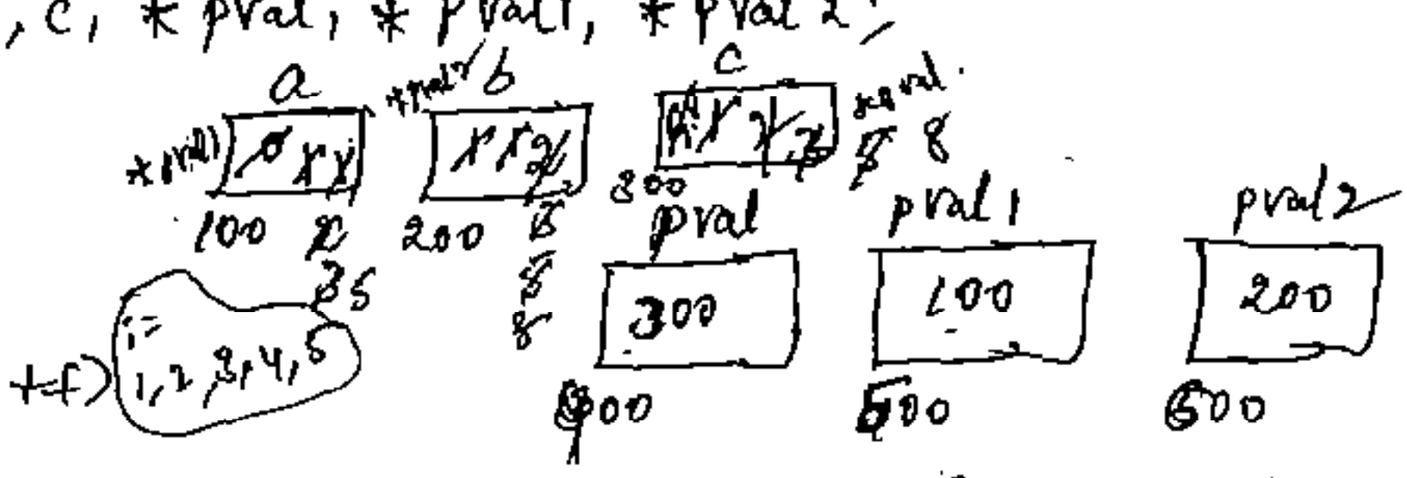
for ex:- int *s; pointer
int **s; pointer to pointer
int ***d; pointer to pointer to pointer.

```
Main()
{
    int a = 10;
    int *p, **p2p, ***p2p2p;
    p = &a;
    (*p) = (*p) + 10;
    p2p = &p
(**p2p) = (**p2p) + 5;
p2p2p = &p2p
Cout << a << *p << **p2p << ***p2p2p;  25, 25, 25, 25.
}
```



→ Method()

```
{ int i, a=0, b=1, c, *pval, *pval1, *pval2;
    pval1 = &a;
    pval2 = &b;
    pval3 = &c;
for(i=1; i<6; i++)
{   *pval = *pval1 + *pval2;
    *pval1 = *pval2;
    *pval2 = *pval;
}
Cout << *pval1 << *pval2 << *pval;  5, 8, 8
}
```




```
Main()
{
  int i=10, *j, *K;
  cout<<i;  10
  cout<<&i;  4004
  cout<<*(&)i;  10

  j=&i;
  cout<<&j;
  cout<<j;
  cout<<*&j;

  K=&j;
  cout<<K<<&K<<*K<<*&K;
  cout<<i<<*&i<<*j<<**&j<<**K<<***&K;

  getch();
}
```

Function in C++:-

def: a Function is an self defined block, which performs a predefined task.

Functions has been defined in 2 ways.

1. Predefined functions. (Library functions)
   @. Ex: printf(), scanf(), sqrt(). defined by the programes.
2. User defined functions.
   defined by the Users.

Ex: Main()

→ Why Main() is call Usee defined function?

The body of the Main() what we provided is depends on User requirement. We can change the every time the body of the Stmt's as per our Requirment.

→ By default any function Nature returns a valee & by default all the functions return type is Integer.

→ We can change the return types of the funcefions as per your requirment, if it is User defined function.

```
Char Main()
{
   printf("%d", sizeof(Main()));  o/p: 1
   getch();
}
```

Void:- Void is a datatype which Kill the nature of a function. the size of void is empty (Nothing).

```
Void Main()
{
   → No return type
}
```

```
Main()
{
   → by default - returns an integer.
}
```

Note: placing the void is Not Compulsary, but when we returning a value to a function we must place some return type, depends on the returnvalue.

```
Main()
{

}     // No Error, but-
```

Warning is came.

i.e "Function Should return a value"

```
Main
{

return 0;
}
```

(or)

```
Void main()
{

}
```

```
Void Main()
{
X
return 0;
}   Error.
```

to overcome this. place return 0. means Program Successfully Completed. (or) use void as return type of main() function., but both void, return, not use.

because every function by default return's an Integer. No one can handle it, that way warning will coming. 'C' is loosely type checking. That's way Warning. C++ has ~~loo~~ raisers an error.

Function Components:- To write the programs with functions On any language we have to ~~know~~ Know the principles of function components.

We have
1. Function declaration (Function prototype)
2. Signature of a function
3. Function calle.
4. Actual arguments.
5. Function definition.
6. Formal palameters.
7. Function callee.
8. return type.

1. Function declaration (or) prototype

```
void Main
{
  clrscr();
}
```

Error: clrscr() have a prototype because predefined function declarations in header files. so we can compulsary add the prog. through header file.

```
#include<conio.h>
Void Main()
{
  clrscr();
}
```

What we write in prog, every variable, function's has a compulsary declare in the prog. predefined functions already it's declarations in header file so, we can add it, the compiler go to header file, & check's the declaration of clrscr() function these already predefined functions already developed by the programmers & kept it in the header files.

Syntax:

Ex: Void clrscr(Void);
↓ Function name. ↳ Signature. ↳ Return type

→ Return-type function-name(arg1, arg2, .. argn); // Function Prototype.

→ The function declaration tell's to the compiler, the Return type of the function, no. of arguments of the function, type of argument's & order of arguments.

→ The function declaration can be defined inside the main() (or) outside, but before compilation. (through header files)

```
int sum(int, int);
↓ O/P   ↓ I/P   ↳ Argument depends on I/P value.
```

The function signature (i.e Arguments) depends on I/P Variables of the program.

The return type of the function always depends on O/P of the program.

The return-type of the function also depends on argument's type.

Ex:- int- Sum (int, int) float- sum (int, float)

long fact (int);

Here 8 no's factorial out-of range of integer. That's way take long as return type. because int- can't hold that value

~~Void numbers(int);~~

Void natural (int); Any series of prog, don't have a return type. So we use Void.

When Multiple values are return, it is not possible to ~~return~~ place multiple return types that's way use Void. because we don't now what is come.

Caller:

The function callee is another component, the caller can present any no of times, any where in the program.

At the time of function callee, there is no need to mension, any return types, function arguments types.

Ex: Void cles(x); X

Actual arguments:

At the time of function callee whatever the arguments we defined, they are called as actual arguments.

The actual arguments can be Variable type, Constant & Expressions.

The actual arguments make a communication to the formal parameters.

Function callee:

Void sum (int, int); // declaration.
↳ Signature

Return type

```
Void sum(int,int);
Void main()
{
  ---
  ---
  ---   → actual arguments.
  sum(10,5); // callee
  sum(10+6, 3+5); // callee
  - - - - -
}
```

→ callee can present - Any no of types.
i.e. More than once also possible

Function callee:

The Function callee is a Separate module, which Contains the definition of of the function.

definition is knowing bal- logic of the module.

→ At the time of function callee, we have to mension Compulsary the Return type of the function.

Void sum(int x, int y) // callee.

Main() is also callee

```
{
  // function definition.
}
```

→ Mandatory as to match the declaration.

At the time of function callee the user explicitly has to mention, the Return type of the function, according to it's declaration.

formal parameters:-

At the time of callee what are parameters, we mentioned they are called as formal parameters.

This formal parameters, will Recive the values from the actual arguments.

The formal parameters must be Variable type Only

```
void Sum (int x, int y) // callee
{
   ------
   ------
}
```

Formal parameters.

Return stmt: The Return stmt will be existed in the Sub mode (Callee). The purpose of the Returnstmt, to make a Communication back to the calla, to which function has been call.

Return stmt can return Only One value

→ The functions which have a Return type has Void can't return a value.

→ A Function programming Caube write in 4 ways.

1. Function with No argumt. & No Return type. ex: clesa();
2. Function with argumt & No Returntype. ex: Void gotoxy(--);
3. Function with argumnt & with return value: ex: getche();
4. Function with No argumnt & Return type ex:- getch();

1. Function No Arg' & No Return value:-

```
#include<iostream.h>
#include<conio.h>
int Void Main()
{
   void draw(void); // declaration
   void Sample(); // declaration
   clrscr();
   draw(); // caller
   cout<<" Main module ... \n";
   Sample(); // caller
   cout<<" Back to main module ... \n";
   draw(); // caller
   getch();
   return 0;
}
```

```
Void draw() // callee
{
   for(int i=0; i<=80; i++)
   cout<<"-";
   cout<<"\n";
} // end of draw

Void Sample() // callee
{
cout<<" It is sub module ";
} // end of sample
```

01/9/ ~~AP~~

Main module
If is SUB module
Back to main module

At the time of function callee it jump's to the sub module i.e function callee.

The program execution always from main() to main() & the Compilation top to bottom.

Tests A function execution always takes place on CPU.

```
#include<iostream.h>
        <conio.h>
Ful- Main()
{
   Clrscr();   //Error
   draw();     - There is No
   ------        function deelaration.
   getch();    - the compiler know's it at compilation time.
  return 0;
}
void draw()
{

}
```

Test2
```
#include<iostream.h
        <conio.h>
~~inf-main()~~
{ void draw()
{

}
inf-main()
{
  clrscr();
  draw();
  ------
  getch();
  return 0;
}
```

We define the definition first- So the compilation already know's about the ~~declaration~~ et callee function name ..

When we define the definition first- there is no need to define any deelarations.

Test 3 #Include <iostream.h>

```
Void draw()
{
→ Return; // Only Control
}
int main()
{
  clrscr();
  draw();
} getche;
  return 0;
}
```

In the draw function we are not passing any value, we just kept the controle.
→ No usage of it.

```
Return; // only control
Return 0; // Control and value.
```

Test 4: #——

```
Void draw()
{ for(i=0; i<=40; i++)
  — — — —
}
int main()
{
  clrscr();
→ int i;
  draw();
  getche;
  return 0;
}
```

Error! Undefined Simbole 'i'.
'i' is local scope.
Were it use, ~~but~~ there only we declare.

Test 5:

```
draw() by default return integer.
{ ——
}
int main()
{ clrscr(); No Error
  int i;
  draw();
getche;
 return 0;
}
```

Test 6:

```
draw(int x)
{
  int x; // Error.
}
int main()
{
  int a;
  draw(a);
  getche;
  return 0;
}
```

Multiple declaration

no need of this declaration

→ Write a prog findout the factorial for a Number.

```
#include <iostream.h>
#include <conio.h>
void fact();
void main()
{
  clrscr(); [for(i=0; i<10; i++)
  fact(); //Callee of upto 10 no's]
}
void fact() //callee
{ int n;
Cout << "Enter the value of n:";
Cin>>n;
long f=1;
  for(; n>=2; n--)
  {
    f= f*n;
Cout<<"factorial :"<<f;
  }
}
```

→ Sum of 2 no's NO Arg's & No return value.

```
#include<iostream.h>
#include<conio.h>
Void Sum(); //declaration
Void Main()
{
  clrscr();
  Sum(); //callee
}
```

```
Void Sum() //Callee
{ int a,b,sum;
Cout<<"Enter the values of a;b:";
Cin>>a>>b;
 sum = a+b;
Cout<<'Sum is:'<<sum;
}
```

Function with Argument & No Return value:

In this catagiry comunication will be takes place from the caller to callee.

There is No Communication from callee to the caller

Ex: fibnocci series depends on the function way of catagiry changes.

```
void fib(int n);
int main()
{ int n;
cout<<"Enter the n value:";
cin>>n;
fib(n);
}
void fibo(int n)
{ if(n==0)
  cout<<n<<endl;
else
   if(n==1)
   cout<<n<<endl;
else
 { int f1=0, f2=1, f;
  cout<<f1<<f2<<endl;
while ((f=f1+f2)<n)
{
    f1=f2;
    f2=f;
  cout<<f<<endl;
}
}
}
```

factorial:

```
#include <iostream.h>
#include <conio.h>
void fact(int); // declaration
int main()
{ int n;
cout<< Enter n value:";
cin>>n;
  fact(n); // caller
} getch();
  return;
```

```
void fact(int) //callee
{
  long f=1;
for(; n>=2; n--)
 { f=f*n;
  cout<<"factorial of n:"<<f;
}
}
```

Test 7

```
Void fact(int);
Main()
{ int n;
  fact();// Error. No Nth Argument.
  getch();
  return 0;
}
Void fact(int n)
{
}
```

fact(int);
Do!- pass type as argument in fact
Only variable name
caller passing the argument to callee.
So pass the variablename.

fact(n);
= In callee we can Specify type & variable.

Advantages of Functions:-

1. By writing in functions debugging of the program is Easy.
2. Function's Support Reusability (Source code).
   i.e Once a function is written it can be called from any other module, without having to Rewrite the Same. This Saves time in Rewriting the Same code.

Sum of two int on 2nd category.

```
Void Sum(int,int); // declaration
Main()
{ int a, b;
  cout<<"Enter the values of a, b";
  cin>>a>>b;
  Sum(a,b); // Caller
  getch();
  return 0;
}
```

Callee

```
Void Sum(int p, int q)
{ int sum;
  sum = p+q;
  cout<<sum;
}
```

→ power logic with Arg's & no returnvalue.

→ Void power (int float, int); // declaration.

```
Main()
{
  int n;
  float a; clrscr();
cout<<"Enter base:";
cin>>a;
cout<<"Enter Exponent:";
cin>>n;
power(a,n); // caller
getch();
return;
}
void power (float a, int n) // callee
{  int i;
   float p=1;
   if(n==0)
   cout<<"value:"<<1;
else
   for(i=1; i<abs(n); i++)
     p = p*a;
cout<<"value is:"<<p;
}
```

abs is used to convert -ve's to +ve's.

3. Function With Arg's & Return value:

In this catagiry: Communication both the sides will be takesplace. The Communication from caller to the callee, will be takes place as arguments & Communication from callee to the caller will be takesplace with the Return stmt's.

→ Write a prg accept a no, display the no in binary formate

```
#Include<iostream.h>
        <conio.h>
        <math.h>
Unsigned long dtb(int);
void main()
{
```

```
int num;
cout<<"Enter num:";
cin>>num;
Unsigned = d to b (num);
long K
{
cout<<"Binary equalent to "<<K<<"
      for the decimal no: "<<num;
   getchar();
}
unsigned long d to b (int n)
{
   unsigned long bin=0, p=0;
   While (n)
   {
     bin = bin + (n%2) * pow (10,p);
         n = n/2;
         p++;
   }
   return bin;
}
```

Memory Map

Instructions	Address	Value
d to b	65516	1001
	65517	
	65518	
num	65514	9
	65515	
K	65516	1001
n	65520	9 4 2 1 0
	65521	
	65522	
bin	65523	0 1
	65524	
	65525	
	65526	
p	27	0 0 0 3
	28	
	29	

bin = 0 + (9%2) * pow(10, 0)
= 0 + (1) * (1) ($10^0 = 1$)
bin = 1

bin = 1 + (1%2) * pow(10, 3)
= 1 + (1 * 1000)
(1 + 1000) = 1001

Factorial using & with Arg & Return value:

```
#include <iostream.h>
#include <conio.h>
long fact(int); // declaration
void main()
{
   int n;
   cout<<"Enter the num:";
   cin>>n;
long c = fact(n); // Callee
cout<<"factorial of number"<<n
      <<" is:"<<K;
   getch();
}
```

```
* long fact(int n) // callee
{
   long f=1;
for(; n>=2; n--)
   f = f*n;
   return f; (or) (; n>=2; f=f*n--)
}

   if (n==0||n==1)
   {
     return 1;
     getch();
   }
```

Whenever a function with return value is present, we can call the function with return stmt (or) without return stmt.

cout << "Factorial of for <<n<<" is:" << fact(n);

Function Calls:

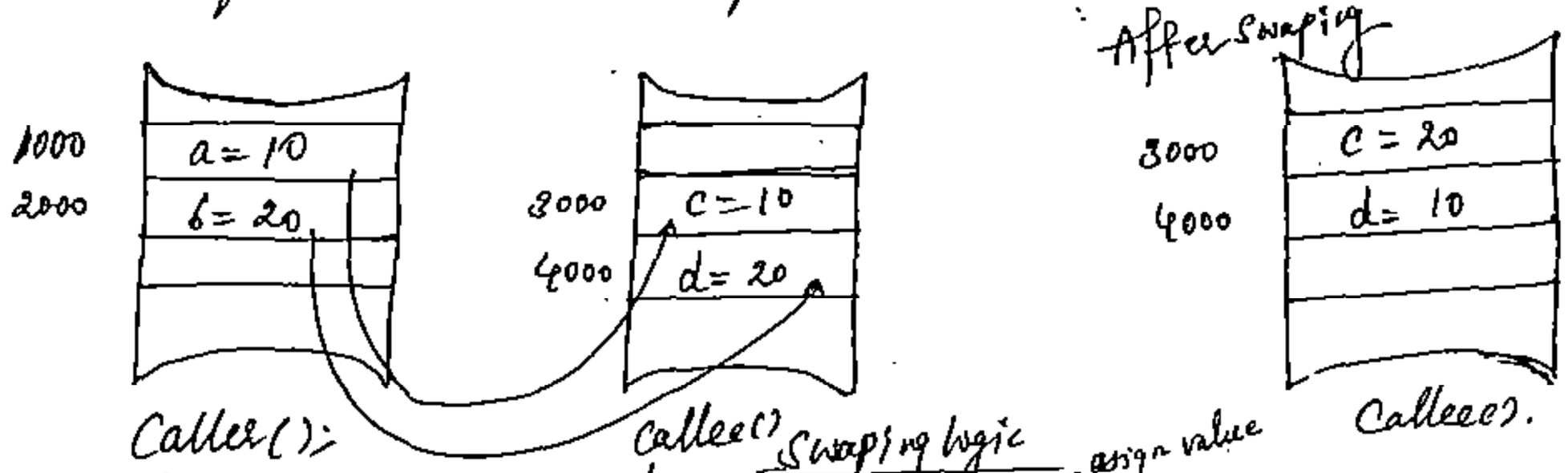
A function calls can be takes place in 3 ways in C++.

1. Call-by-value
2. Call-by-address
3. Call-by-reference (New in C++)

Call-by-value:

Sending the value of a variable from caller to the callee is called as Call-by-value.

If any changes made in the sub function it will not effect to the calle function variable values



Swaping logic

d = (c+d) - (c=d) (or) a^=b^=a^=b;

d = (10+20) - (c=20)

d = (10+20) - 20

d = 10

^ = Bitwise exclusive or

```
#include<iostream.h>
<conio.h>
void swap(int,int);
void main()
{
  int a,b;
  cout<<"Enter the values of a,b:";
  cin>>a>>b;
  swap(a,b);
  cout<<"After swaping:"<<a<<b;
}getch();
```

```
void swap(int a, int b)
{
  b=(a+b)-(a=b);
  cout<<"after swaping:"<<a<<b;
}
```

$a \wedge = b \wedge = a \wedge = b$

$a = 10$ R.H.L

$b = 5$

$a \wedge = b$

$a = a \wedge b$

1010

0101

$a = 1111$

a	b	a ∧ b
0	0	0
0	1	1
1	0	1
1	1	0

$b \wedge = a$ $b = b \wedge a$

5

0101

1111

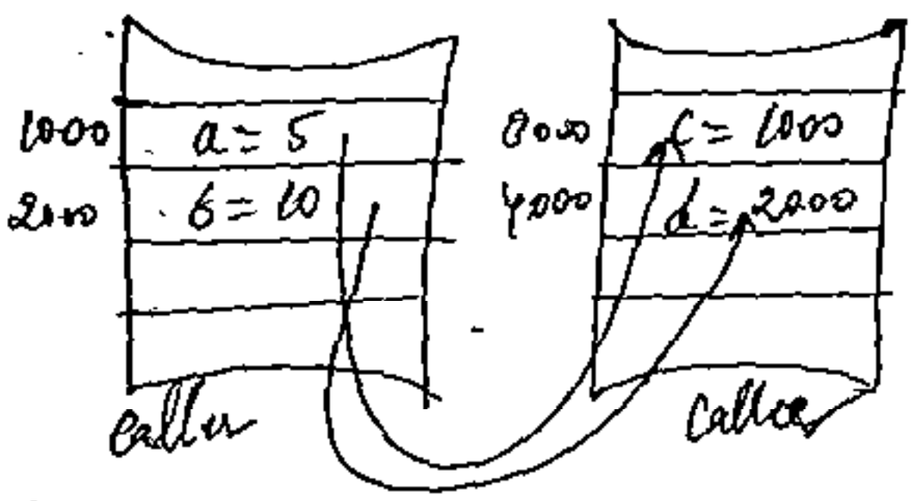
$b = 1010$ Continues.

like this

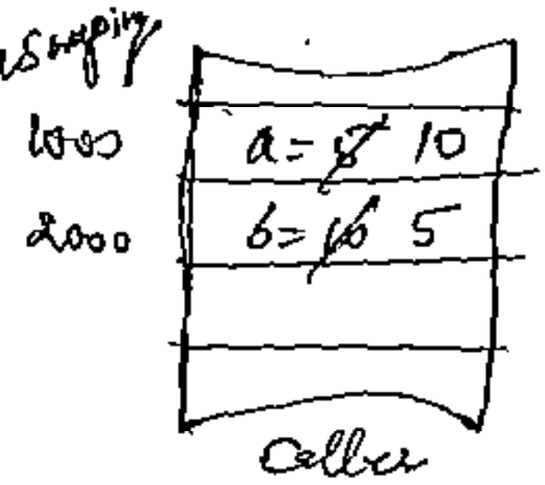
Call-by-address:

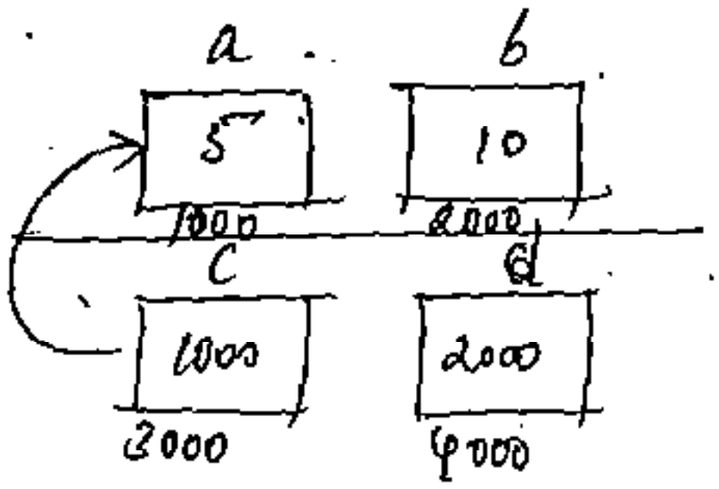
Sending the address of the Variable from the caller to the callee is called as Call-by-address.

If any changes made in the subfunction, it will effect to the caller function Variable values.



After swaping





*c = a => 5

*d = b => 10

logic

*d = (*c + *d) - (*c = *d)

```
#include <iostream>

void main()
{
    void swap(int *, int *);
clrscr();
int a, b;
cout<<"Enter the values:";
cin>>a>>b;
swap(&a, &b);
cout<<"After swaping in caller function ()"
    <<"a:"<<a<<"b:"<<b<<endl;
}
```

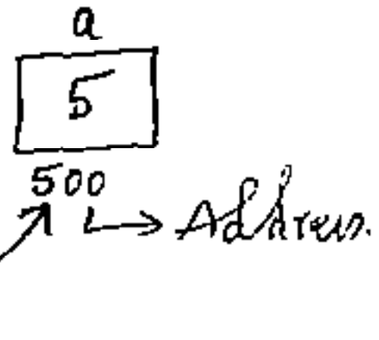
```
void swap(int *c, int *d)
{
 *d = (*c + *d) - (*c = *d)
cout<<"after swaping in sub
    fun()"<<"c:"<<*c<<"*d:
 "<<*d<<endl;
}
```

Reference Variable:

Syntax: data-type & reference-variable = Value_Variable;

int a = 5;

int & z = a;



a, z point's to same location value.

A reference variable look's like an ordinary variable but-act's as an pointer variable.

a reference variable will be indicated by '&'

a reference can be treated as an as alias. (duplicate)

→ Every reference variable, must-initialise at the time of declaration.

→ a reference variable can be created to an integers, float, characters & double.

It is also possible to refer ADT (Abstract data type) (User define).

Ex:-

```
Char ch = 'A';
Char & ch1 = ch;

float y = 45.68;
float & s = y;
```

```
int b;
int & a = b;

double length;
double & height = length;
```

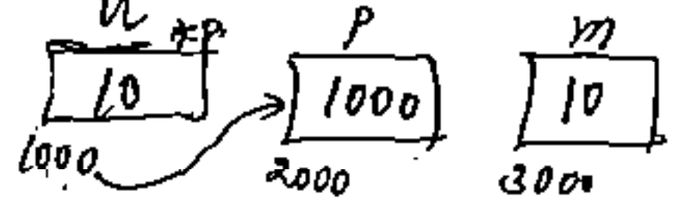
→

```
int y[100];
int & x = y[5]; // x is an alias y[5] element.
```

Here only One Element- i.e the location y[5].
i.e 6th Element- effects. with x.

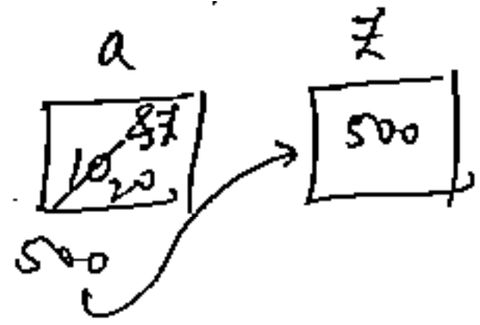
→

```
int n = 10;
int *p = &n;
int & m = *p;
```



Here Any one Changes all with Variable will effect-

```
#include <iostream.h>
          <conio.h>
Void main()
{ int a=10, b=20, c=30;
  int &z=a;
  clrscr();
  cout<<"A:"<<a<<"B:"<<b<<"C:<<"c"z:"<<z<<endl;
  z=b;
cout<<"A:"<<a<<"B:"<<b<<"C:"<<c<<"Z:"<<z<<endl;
cout<<"Address of a:"<<&a<<endl
    <<"Address of b:"<<&b<<endl
    <<"Address of c:"<<&c<<endl
    <<"address of z:"<<&z<<endl
```



10,20,30,10

20,20,30,20

O/P:
a, z, address are same Here.

Note:

→ a reference variable can refer to only one variable at a time.

Ex:-
```
int &z=a; // Valid
int &z=b; // Invalid.
```

→ a Variable can have Multiple references, hence Change in one. if changes all.

Ex:-
```
int &z=a; // Valid
int &s=a; // Valid
int &k=a; // Valid.
```

→ A reference concept can we called as pointers, but not pointers. (i.e act like pointers)

Differences b/w Pointers & Reference Variables

1. a pointer variable can initialise, any no of times. but a reference variable can initialise only one time. due to this it can be called as Constant pointer

2. pointer variables can initialize any time & any no. of times, & the pointer variable get automatically dereferenced (changes)

3. The pointer's will accepts Separate Memory.

4. The pointer variables it's not-so <u>eligant</u>-(<u>Clear</u>) When we Working with pointers.

5. No Security.

2. a Reference variable can be initialise Only one time, at the time of declaration only.

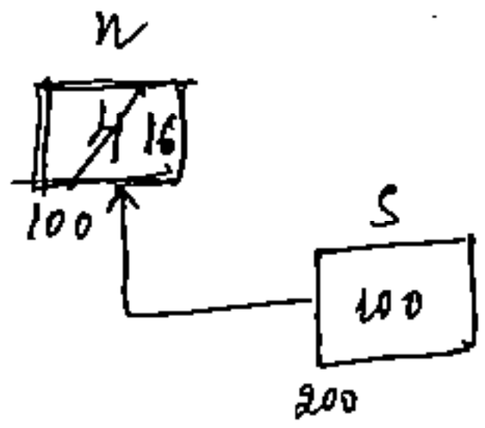
3. The reference variable will point-out Same memory of value variable.

4. The Reference variable is eligant & easy to implement for the user.

5. Secure than pointer's.

<u>Call-by-Reference</u>: <u>Sending the</u> ~~address~~ Value <u>of the variable through Cross linkage & Receiving the address of the Variable is called as Call-by-reference.</u>

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

Void DoubleNum(Inf &);
Void Main()
{ Inf n;
 Cout<<" Enter the n:"; 4
 Cin>>n;
 DoubleNum(n);
 Cout<<n; 16
}
Void DoubleNum(Inf &S)
{
 S=S*S;        4*4=16
 Cout<<S<<endl; 16.
}
```



&S = 100
S = 4
S*S = 16

Advantages of References:

1. When we passing a variable as a parameter for the functions better to implement by call-by-reference.

When we intent to change the value's of actual argument through the called function.

→ It is better to save memory by preventing the creation of large set of variable's that are being passed to the function

→ All ways implement reference concept, since save coding with reference.

→ Unsave coding with pointers.

→ When reference variable is lost (any one accessed that data) only that value is lost, but if the pointer is lost, the ~~to~~ whole program security comes into the question.

Applications:

The internet architecture is developed by Ctf. under call-by-reference concept. and many other applications.

shown in Chat dialog box ☐ Observe the gtalk Chat box.

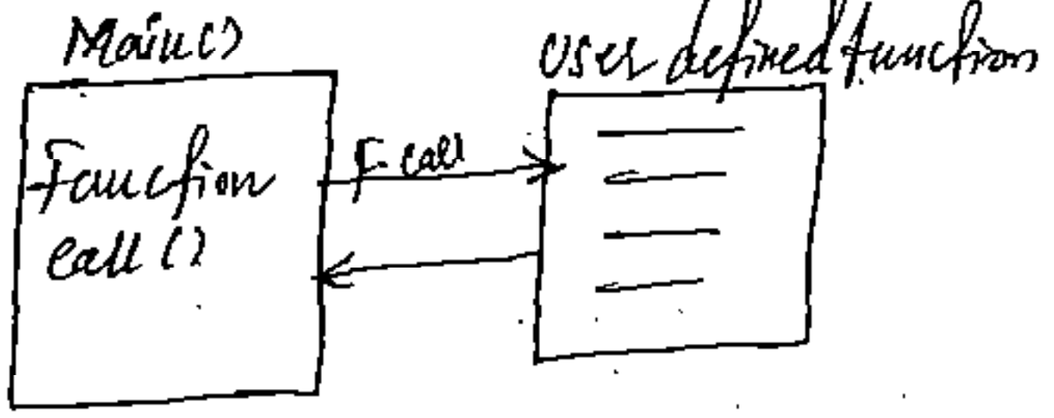
~~Observe~~ used in ~~radio~~ video conferencing, teliconferencing, live telecast etc

Diff b/w Call-by-references, call-by-value, call-by-address

Reference	Value	Address
1. Sending a variable to function is Call-by-reference.	Value is send to the function.	Address is send to function.
2. Formal parameters are reference variables i.e int &x, int &y.	2. formal parameters are normal variables i.e int x, int y	

2. formal parameters are pointer variables. i.e int *x, int *y.		
3. actual parameters must be variables. i.e Change(a,b);	- Actual parameters can be value variables, constants, @) expressions. Chang(a,b); Change(5,10); Change(a+f, a+5);	Actual parameters must be addresses. i.e Change(&a, &b)
4. Changes made to the formal parameters are reflected to actual parameters.	Not reflected.	Reflected.
5 The Actual parameters & the formal parameters address will be same.	Different values.	Address are also different.

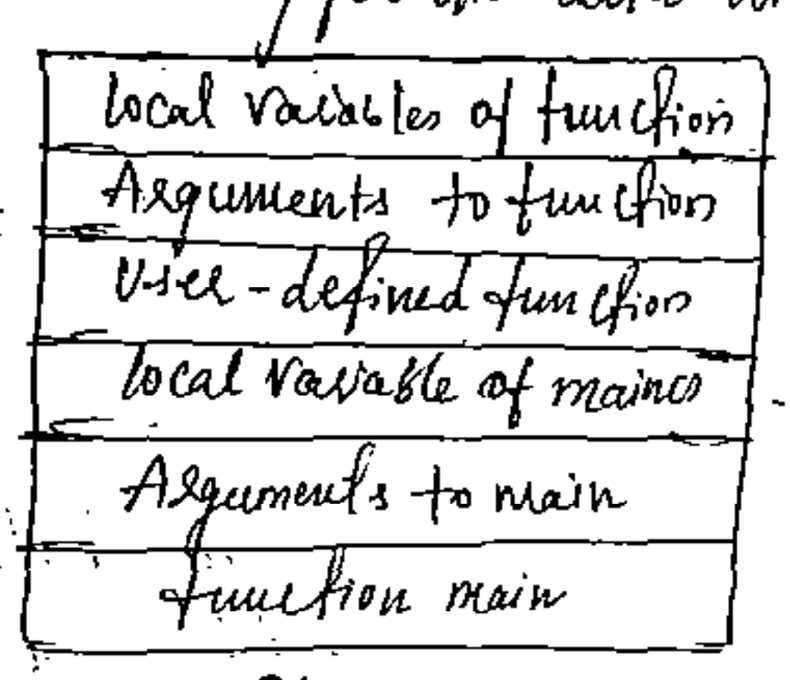
How function works:-



a function execution always takesplace on cpu & when the function is executing it follows the Run-time Stack desiplain.

At the time of functioncall every function data will Store On Stack desiplain.

Function Calls & Stack



→ For the single function call all the total decriptain of storing the arguments, local variables. This willbe takes place to implement this decriptain the performence of the system willbe degrade.

In 'C' lang we can't reduce this burden, if the function definition contains, more no. of stmt's (or) less no. of stmt's.

When a function call is taken place it is Navinating (Moving) to the external scope & comming back to the program. (stack decriptain)

Inorder to reduce this burden we can use inline() function

definition of inline():-

→ The function code get's inserted, instead of a jump to the function, at the place where there is a function call is present, is called as inline function.

→ When we define inline functions, we request the compiler to insert the code at the place where there is a function call. So that time is saved in calling the function & returning from the function.

→ If a function is declared as inline at the time of compilation the body of the function is expanded, at the point at which it is invoked. (call)

→ For small functions the inline functions provide modularity & removes all the over heads (burdens) of the function calls.

→ The inline specifier instruct the compiler to replace the function with the code of the function body. This substitution is called as inline expansion.

→ The inline keyword tell's the compiler prefer inline expansion instead of normal function calls.

When the definition contains loops & Recursion Concept-
inline does't work's. That means if it doesn't raises any error
normal function expansion will be takeplace. It may raises
warning

→ inline will be defined at the time of definition (or)
we can define at the time of prototype. (definition first write in prg)

Example on inline:

↳ definition.

```
inline int max(int a, int b)//caller
{ return a>b ? a:b; // before ? is satisfied then it returns
}                      before-satisfied value, otherwise
                       after ?
Main()
{
 Cout << max(10,20);//caller
 Cout <<" "<< max(99,88);//caller
}
```

Internally

```
Cout<<a>b?a:b;
Cout<<" "<<a>b?a:b;
```

like this replacement is done for every caller function. So memory occupancy increases

→ Due to inline function the memory can be increase.
Inline functions, are functionally similar to #define Macro's.
In both the cases during compilation time - the body of the function (or) Macro is expanded.

But inline functions are preferred over Macro's.

→
```
#include <iostream.h>
#include <conio.h>
#define Square(x) x*x
 Void main()
{
 clrscr();
Cout<< Square(1+2)<<endl;
 getch();
}
```

(1+2) substitute in the place of Square(x)

So x*x = 1+2 * 1+2
(1st reference)

2*1 = 2

1+2+2 = 5

5
```
#Include <iostream.h>
#include < Conio. h>
# define Square (x)  * X * X
Inline Inl- Square (Inf x)
{
 return X * X;
}
 Void Main()
{
 Clrscr();
 cout- <<" Square (1+2) <<endl; 9
getch();
}
```

In the above program the Macro display the o/p : 5. because all the occurences of Square(x) are replaced with X * X. So in this case Square (1+2) is Replaced with 1+2 * 1+2.

1+2 +2 = 5

When we defines Inline functions. the parameter expression is evaluated first & then send to the function. So the function body will evaluate the expression as 3 *3 =9.

When both macro, & Inline ~~are both~~ are present in the program, Preference to inline only. Macro not evaluted

Difference b/w Normal functions & Inline functions.

Normal:	Inline:
1. Function call leads to branching i-e Controle branches to the called function executes & controle returns to the calling function.	1. Function call's to leads to substitute. i.e function call is replace & Code is substituted
2. Slow in execution, Since branching is involved.	2. Fast- in Execution, Since branching is eliminated (Jumping)

3. Generally function prototype existed.

4. There can be any no. of Stmt's including Control structures.

5. It is proceded by CPU.

2. This doesn't exist function prototype.

4. It must be a Small function without any Control Structures.

5. It is proceded by Compiler.

Diff b/w Macro's & inline functions

Macro's	Inline
1. Macro's leads to substitute.	1. Inline also leads to substitute
2. It is a preprocessor which expands Macro's.	2. It is a Compiler which substitute the function code replacing function call.
3. Proceded by Preprocessor.	3. proceded by Compiler
4. It can contain any no. of Stmt's, including Control Structures.	4. It contain less no. of Stmt's without control Structures.
5. Substitute takesplace before Compilation.	5. Substitute takesplace during Compilation.
6. Macro's can't return a value	6. Inline function can return a value.

Function Overloading:

C++ supports a new concept of functions called as ~~Oper~~ function overloading

def: defining multiple function names with the same name, is called as function Overloading.

Overloading is not the Concept of C++, it supports Overloading despite.

→ Overloading is the methodology of polymorphism & it is a Conconcept of Oop's. Not a Cft.

→ The Advantage of overloading, it is an Enhanced flexible. it never reduce the definition.

Generally function reduce the Performence of the System & by the function overloading, it takes Somewhat more.

→ Overloading can be takesplace not only functions, but for the Operator's also.

→ When function overloading is takesplace is Only for the User convinience for checking of the functions it is issy to recognise very large functions by the function overloading Concept.

→ When function overloading is taking place we have to be defee by 1. no. of arguments.
2. Order of arguments.
3. Atleast type of argument's should be different when overloading takerplace

Ex: → Sum (int, int);
Sum (float, float); // Type of arg's are diff
Sum (float, int);
Sum(int, float); // order of arg's are diff

→ Sum(int, int);
Sum (int, int, int); // no. of arguments are diff

```
// Example on function Overloading.
  #include <iostream.h>
  #include <conio.h>
  #include <string.h>
  inline int add(int x, int y).
{ return x+y;
}
 inline float add (float x, float y)
 { return x+y;
 }
 inline int add(int x, int y, int z)
 { return x+y+z;
 }
 inline double add (double x, double y)
 { return x+y;
 }
 char * add (char *x, char *y)
 { return (strcat(x,y));
 }
  void main()
  { clrscr()
  cout << add (12,34,56) << endl;
  cout << add(45,567,234.45) << endl;
  cout << add(10,20) << endl;
  cout << add ("hydera", "bad") << endl;
  cout << add (89.3453f, 67.76f);
   getch();
}
```

→ The Compiler Never Recognise by the return types.
if is possible only by arguments.

9

Applications of Overloading:

1. In real time the scientific calculator's nearly 24 to 36 functions are overloaded.
2. The computer keyboard is overloaded.
3. The Cellphones has been overloaded.

due to this if decrease the Size of the Machine

In Machine Oriented programs (Embeded systems) function overloading mostly used.

4. In Database function overloading concept is on Select-Command.

5 Mouseclick is one event, the event is converted into functions.

Notes

→ Function Overloading externally depends on the arg's of the function, but Internally Some other decipher will be takes place

→ C++ Compiler Changes name of all the functions definitions & calls while compiling the program this is known as Mangling.

(Imp) Mangling: Mangling is a process different names are given to the different versions of Overloaded functions.

Ex:-
```
Void f(int i)
{
}
Void f(float f)
{
}
```

```
Void f(float f, float f)
{
}
  Void main()
{ f(10);
  f(3.5f);
} f(10,3.5f);
```

→ The Compiler changes the names of the overloaded function as follows.

Original function name:	Mangled Function name
Void f(int i)	?f@@YAXH@Z(int i)
Void f(float f)	?f@@YAXM@Z(float f)
Void f(int i, float f)	?f@@YAXHM@Z(int i, float f)

→ This formate internally done by the Compiler

→ Name mangling also known as name "decoration".

→ Name mangling is Compiler dependent. different compiler may mangle the same function name differently, compiler mangles the name as per the type, order and no. of arg's.

Default Arguments:

→ C++ supports one more extra concept i.e "default arguments."

An argument value that is specified at the time of definition (or) at the time of function prototype, is called as default args.

→ This value automatically pass to the function parameter when no explicit argument is specified in the function call.

→ Default Arguments are takes place in the function prototype (or) at the time of function definition.

→ The default arguments whenever implemented, it has to be defined from right to left, to the leftmost argument.

→ Default arg's are very useful while making a function call

at the time of the user not passing any type of values.

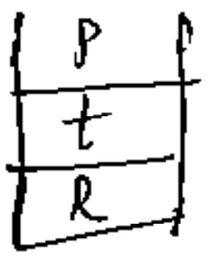
(or) Same type of values are existed.

→ When you passing Original Values, the default values will not takes place into the picture.

Bank (int- P = 5000, int- t = 2, float R = 0.15) ✓

Bank (int- P , int- t, float R = 0.15) ✓ ←

Bank (int P = 1000, int t, ~~float~~ R = 0.15) X



Error

Bank ();

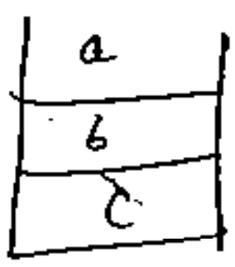
Bank (6000);

Bank (6000, 5.9);

→ Missing the Right-most parameter, defining deft most- if raises an error.

// Example on function Overloading.

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

inline void sum( int- a=100, int- b=200, int- c=300)
{
 cout- << a+b+c <<endl;
}
void main()
{
 clrscr();
 sum(10,20,30): 60
 sum(10,20): (a=10, b=20, c=300) 330
 sum(10): 510 (a=10) b=200) c=300)
 sum(): 600 (a=100, b=200, c=300)
 get-ch();
}
```



argument- top to bottom

, str the variables are inserted into the stack. after from top to bottom called values are assigned

Applications:

→ By using default arguments some of the applications booting of a system (Checking), Saving a file, Shutdowning the system. This type of applications will be takes place.

→ Edit plus is designed by default arg's.

The advantage Memory Saving, Time Saving & User Consienence.

Ex:-
```
void fn(int x)
{
  cout<<x;
}
void fn(int x=10, int y=20)
{
  cout<<x<<y;
}
main()
{
 int a=1, b=2;
   fn(a);
}
```

→ When this program is executed there is a changes of ambiguity. i.e Confussion for the compiler we function has to invoke, because both the way of defining the definitions, is the right ~~write~~ only. Since C++ supports function overloading & default arg's.

→ Not to get this ambiguity remove any one of the definitions.

Q. what are the diff b/w 'C' and C++ languages? (Language wise diff)

'C'	'C++'
1. Operator's are treated as Special Simboles, with Uniq Capable task in 'C' lang.	1. The operators are treated as Special functions which are abstracted with defined simboles.
2. Under 'C' compiler the External printf(); Library files are included with the 'Main Objal'. with altimatly increase the size of the program.	2. The 'C++' compiler navigates b/w the Source code & the Library files to execute the External file Separatly.

3. The functions are evaluated on partial signature.

4. The source code can be represented only in the form of functions.

5. 'C' Compiler have a specific declaration block.

6. 'C' Compiler depends on functional Structures. The ~~evaluation of~~

7 Under 'C' Compiler the functions can be declared at declaration block. which is not mandatory

8 Under the 'C' Library files Contains the declarations of the functions where as the definitions are placed in Cos.obj (available in Library (Lib) folder)

3' The evaluations of the functions can be taken place by considering the complete signature

4. The source code can be represented only in the form of objects.

5' C++ Compiler doesn't have any specific declaration block in which the variables can be created in various locations.

6. The evaluation of data access mainly depends on Operator's along with Objects.
ex: cout << operator
obj.

7 C++ Compiler required forward declarations.

8 The Library files under C++ includes the declaration, & partially the definitions (logic)

Dynamic memory allocation:-

The static memory allocation like arrays. there is a wastage of memory blocks, at the time of execution.

To overcome this problem ~~where~~ we are implementing Dynamic memory Concept.

The Advantage of dynamic memory Concept to allocate the memory at the time of execution, without any wastage of memory block.

In C++ the Dynamic memory concept is supported by new & delete operators.

→ The memory allocation will be takesplace by the 'new' operator and deallocation will be takesplace by 'delete' operator.

Memory model in RAM

Stack Area
Heap Area
PMA Area

(free store).

→ The Dynamic memory allocation will take place using One of the memory area called as Heaparea.

Heaparea in → Datasegment in → RAM.

→ To allocate the memory dynamically use the 'new' operator.
The 'new' operator returns a pointer to a allocated memory.

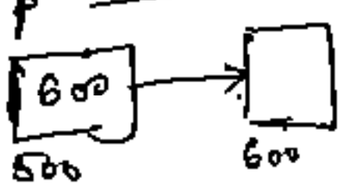
syntax:-

datatype * var_name = new datatype [size];

→ After defining some size, the base address assign to the pointer.

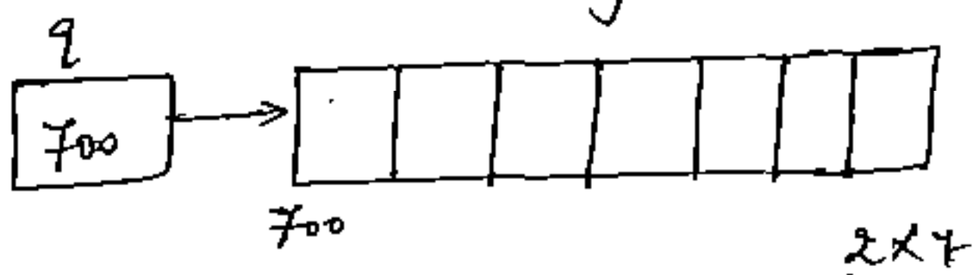
a pointer is the address of the first byte of the memory.

for ex:-



int * p = new int; // allocate one uninitialised int
// int * means pointer to int

int * q = new int[7]; // allocates 7 ~~units~~ Uninitialised ints
an array of 7 ints



2 x 7 = 14 bytes

int

The 'p' doesn't no how many elements if is pointing. (i.e Pointer variable). The 'p' doesn't no what type if is pointing.

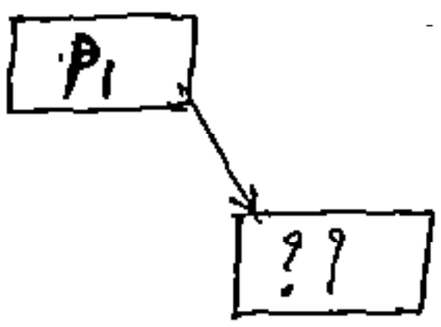
double *pd = new double[n]; // allocate uninitialised doubles

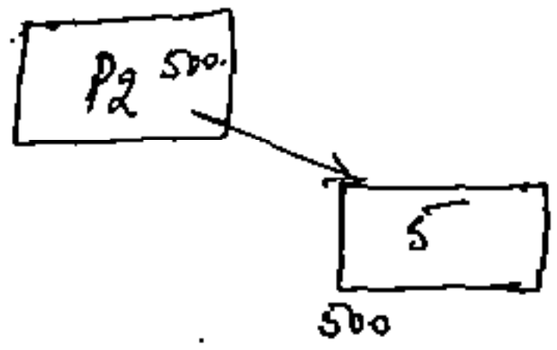
→ This memory allocation will be takes place at the time of execution.

At the time of declaration we can also initialise the elements.

→ int *p1 = new int; get a new (allocate) uninitialised int

→ int *p2 = new int(5); // get a new int initialised to 5.





→ int *p3 = new int[5]; // gets (allocate) 5 ints.

// array elements are numbered 0, 1, 2, ...

→ ~~p[3]~~ p3[0] = 7; // write to ('0th) the 1st element of p3

p3[1] = 9;

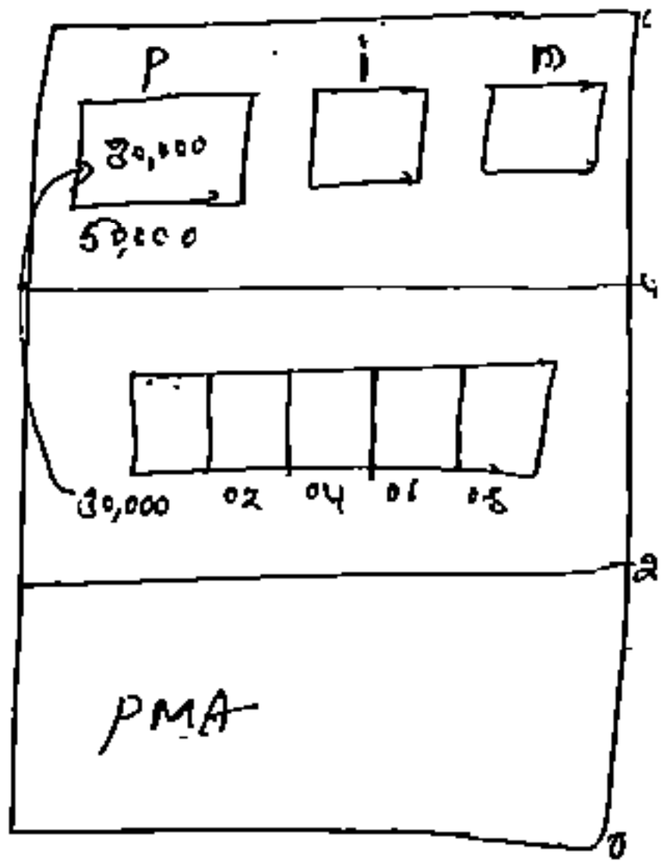
→ Write a program accept the memory dynamically store some data & display them.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
  clrscr();
int *p, i, n;
```

```
cout<<" Enter the size of n :";
cin>>n;
if(p==NULL)
{
  cout<<" Memory allocation Failed....";
  getch();
  exit(0);
}
cout<<" Enta the elements :"<<n;
  for(i=0; i<n; i++)
  {
  cin>>*(p+i);
 cout<<" the Elements are .."<<n;
  for(i=0; i<n; i++)
  cout<<*(p+i)<<" ";
  delete []p; // deallocation an array
  getch();
}
```

Stack area

Heap area (freestore)



if do like this delete p; only one element is deleted. but we want to deallocate total array. delete []p;

```
int *pi = new int; // default initialisation
char *pc = new char('a'); // explicit initialisation
double *pd = new double[10]; // allocation of (uninitialised) array
```

⇒ delete and delete [] return the memory of the an Object allocated by new to the freestore, so that the free store can use it for new allocations.

```
delete pi; // deallocate an individual object.
delete pc; // deallocate an individual object.
delete []pd; // deallocate an array
```

17 → Delete of Zero-valued pointer ("The null pointer") does nothing.

```
Char *p=0;
delete p; // harmless.
```

Note: In dynamic memory concept deallocation will not takesplace automatically, we have to explicitly have to deallocate after the program is finished.

The deallocation in C++ will be takesplace only by the "delete" operator.

→ The differences b/w the malloc() function & new operator.

→ The new operator is a simple syntax comparing with malloc() function.

The new operator doesn't need to perform any explicit type casting. Implicitly type casting will be takesplace.

Object-Oriented Programming

Q: Why do we require an object-oriented programming?

A) For the reasons of 1. Modulaity } From functions.
2. Reusability
3. Security
4. Extensibility
5. Efficiency
6. Simplicity

We achive above six reasons things by

1. Encapsulation.
2. Inheritance.
3. polymorphism.

def: of OOP's:

✓ The programming in which data is logically represented in the form of a class & physically represented in the form of an object. is called as object-oriented programming

(or)

A systematical approch which having a tendency to Stablise the programming structures with Complete Security & durability

→ We have Object-Oriented Programming, Object-based programming
The differences for object-oriented programming consist of
Class, object, data abstraction, Encapsulation, Inheritence, Polymorphism, dynamic binding & Message passing
ex: C++, Java
The object-based programming doesn't support's Inheritence & dynamic binding Ex:- JavaScript, Visual Basic (Vb 6.0). 31

Encapsulation: The Encapsulation placing the data & the functions that work on the data in the same place.

While working working with procedural language if is always not-clear, which functions works on which variables but object-oriented programming provides you a framework to place the data & the related functions together in the same object.

def: of Encapsulation: Wraping of data members & member functions in a single unit is called as encapsulation.

The encapsulation hides all the deteiles of an object.

The advantage of encapsulation.

1. Data hiding
2. Binding of Grouping-

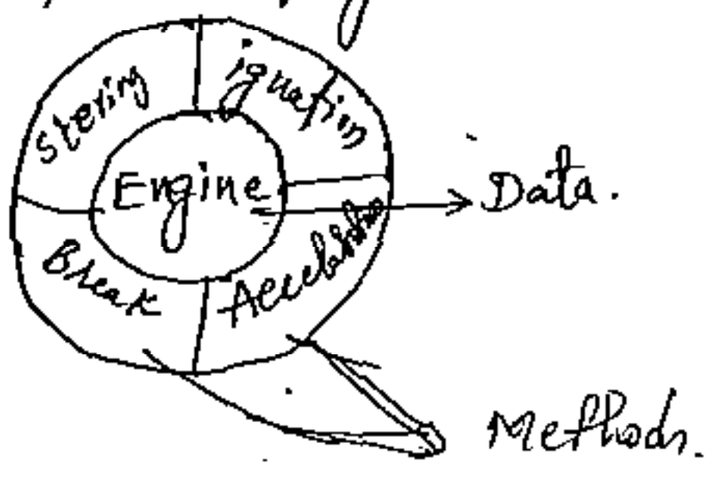
Q. What is data hiding?

A) Securing data from being accessed from other parts of the program is called datahiding.

Ex:- If you take an application like CAR we implement many operations like ignition (starting), acceleration, breaking, etc.

This all will be takes place the drawing of the car with out the knowledge of the Engine

So the Engin is a data, we not accesing directly. This type of concept helps the programmer to Implement Secure program.



Data abstraction: Data abstraction refers to providing only essential information to the outside world & hiding the background details to represent the needed information in program without presenting the details.

Ex:-

a Database system hides the details how the data is stored & created & maintained. Similarly the way of C++ Class provides diff methods to the outside world without giving internal details out about those methods & data.

→ When Encapsulation is present by default abstraction is created.

def: of Data abstraction: It is a process of identifying properties & methods related to a perticular entity as related to the applications.

(or)

Data abstraction is a process of showing the essential details without showing the background details is called as data abstraction.

Ex:- <istream.h>

cout.operator<<("hai");

cout.operator<<(5);

<< is also a function

Class

a class is an user defined datatype, when you define a class you define a blueprint for an object.

All the Reserved rides

→ a class is encapsulated with data & related operations.

A class is a Collection of Members

1. Data Members.
2. Member functions. (Method).

→ Class is a blueprint (or) logical view of object. (like Student).

→ class is a plan for before constructing an object.
object can also access methods only. not - Data members.

→ a class is a reusable program (or) module.

→ a class is an ~~theatrical explanation~~. theoritical Explanation.
an object is an participating entity

→ Class is a theorical explaination for a particular type of data flow (or) behaviour each Stream is associated with one class.
behaviour (or)
flow of data.

Class → Stream → behaviour → object

Class is collection of methods.

Note:

→ The class is the "fostream" in C++. In C++ each file is one object of a peticular class. every stream should be part of one class. every class should be part of a file.

→ Class Supports Object-oriented programming system not Object-oriented language

→ C++ provides the fesility for object-oriented, the class helps in hiding the data & functions from extranal use.

→ The data members of the class can be called as attributes.

the attributes always satisfy the domain.

Oops theory always concentrates on semantics (logical things)

→ class student
{
int sno;
float fee;
};

// this are attributes

→ class customer
{
int sno;
float fee;
};

// this are not-

attributes & domain has a relation ship. Customer, class & attributes are not matching.

Attributes are those indicates stable features.

→ Attributes always should have binary relation b/w a class and certain domain.

syntax of class:

class <class-name>
{
<visibility-match>:
data members 1;
:
datamembers n;
<visibility-mode>:
Member function 1;
Member function 2;
- - - - -
};

→ The c++ class can be ended with ; (semicolon). The semicolon indicates, it is end of the class. That means we can define global variables.

The Semicolon (;) indicates my scope of a class is finished.

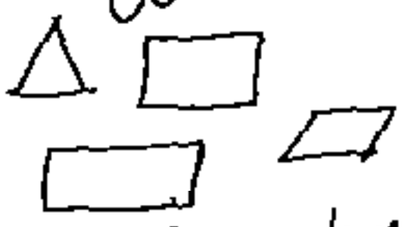
A class can also be defined if it is a group of objects, that have the same properties, common behaviour, common relationship, ex:

if we accepted polygon class it has some properties Vertices, boarder colour, fill colour,

and also has some methods (member functions)

Draw(), Erase(), move(), etc.

Polygons:



each one have common behaviour & relationships.

polygon class is a group of objects.

entity = living (ex) Non living being

→ A characteristics required of an object (or) entity when represented in a class is called as property. (or) attribute (or) characteristic.

→ An action required of an object (or) entity represented in a class is called as object.

action is knowing but a method.

// Designing a class named ~~dog~~ dog

```
class dog
{
  curved tail;
  Carnivorous teeth;
  hair size;
  color; gender; height; weight; length;
  eating();
  barking();
  running();
  mating();
  begging();
  biting();
  seeing();
};
```

Objects Streetdogs, pamalian;

Street-dogs(yes, yes, 1cm, brown, m, 1.5ft, 20kg, 3ft);

Objet: An Objet is an entity that have a physical boundary. It can be called as participating entity. It represents an entity in the real world.

There can be physical objects, computer user environment, user defined datatypes.

An Objet must be self initialised & self intiligence nature.

Every Objet is fixing it to it's boundary (or) else objet will misbehave.

a real world objet should have 3 charecteristics (or) it has to satisfy 3 things

1. State.
2. Behaviour.
3. Identity

if the 3 things are satisfied it is called real Objet.

→ "State" means properties (or) attributes (or) charecteristics.

Some properties Visible & Unvisible

Ex: Mango is a fruit class, colour is visible but taste not visible

→ By using properties we can able to provide the actions this actions are known as behaviours.

Dependending on the State Coresponding behaviour is existing When there is no state, there is no behaviour

State & behaviour both are interdependent.

→ Identity means a name is given inorder to identify the object.

Object-

2nd def: An Object is a combination of set of data values as datamembers & set of Operations as Member functions.

cout.operator<<(5);

↓ (dot) is Operator.

↳ Insertion operator is a Member function.

In Off Datamembers are ~~called~~ call with help of operator operator & member function. @ method.

→ Once a class is Created, we can create any no. of objects.

Each Object has it's own properties (or) Charecteristics. that describes what it is.

for ex: properties of a person object

Name
age
weight

properties are knowthing as data members.

→ Some properties of a CAR color, weight, model no, no of wheels, engin power.

→ An object also executes some actions, the actions of a CAR ~~Start, Stall~~ Stop(), Start(), Accelelate(), Reverse();

→ The Match b/w programming objects & realworld object's is the Result of Combining the properties & the actions of an Object.

An Object is an Conceptional entity that can be identify.

→ An Object & Object will communicate, when only State, behaviour, & identity of other object is provided.

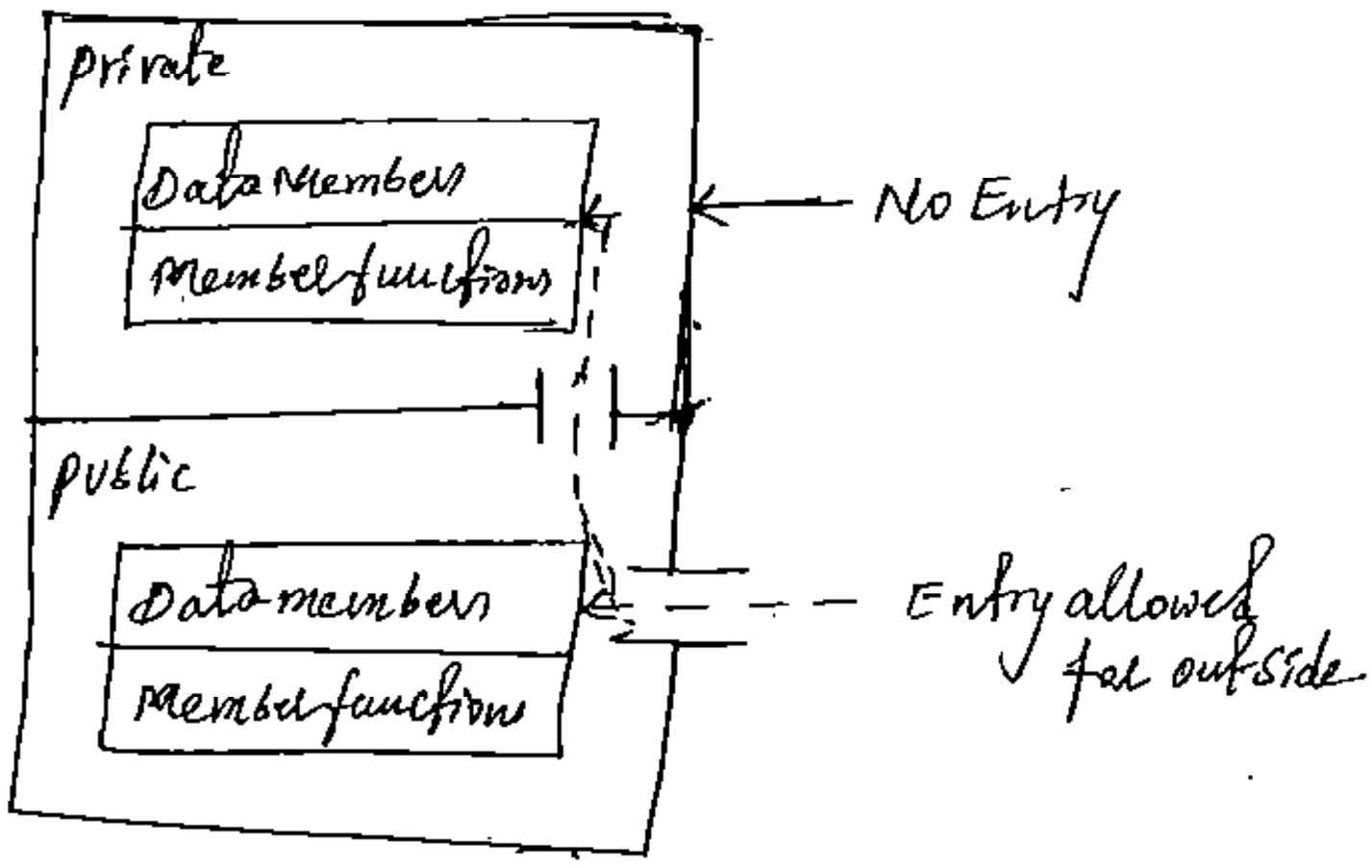
→ Every object is a Conceptional entity when boundary is Changing it's Conceptional entity has to change.

EX:- ~~Right~~ rite now we are Students.

Class-Room → Student
↓
boundary
Home → Family person.
Hotel → Customer.
} Conceptional Entity.

→ One object never has one behaviour, we can't know that behaviour of object, when the object is only one boundary

Ex: Mouse.



As in the class programming of c++ Supports 3 types of visibility modes

1. Private
2. Public and
3. Protected.

a class by default is private These are called as visibility modes.

→ Create a class of student:

```
#include<iostream.h>
#include<conio.h>
class student
{
  private: int sid;
           char sname[20];
           float fees;
  public: void getdata()
          {
```

```
cout<<"Enter the sid:";
cin>>sid;
cout<<"Enter the sname:";
cin>>sname;
cout<<"Enter the fees:";
cin>>fees;
}
void putdata()
{
cout<<"sid:"<<sid<<endl;
cout<<"sname:"<<sname<<endl;
cout<<"fees:"<<fees<<endl;
};}
void main()
{
classes;          → User defined - datatype.
student s;        → object
cout<<"size of object s:"<<sizeof(s)<<endl;
s.getdata(); // function called
s.putdata(); //   → Member functions.
getch();
} // end of 's'
```

When Main() will closed objects file will be ended.

In a program
size of object = size of Data members.

→ ~~Why~~ Data members stored in object.

→ Why stored data in Object. means, object have the Memory.

Test 1:

```
class student
{
    int sid;
    char sname[20];
    float fees;
};
void main()
{
classes c;
student s;
s.sid=234;
getch();
}
```

O/P Error

O/P error

Since we can't access the abstracted part of data from outside the class.

→ Since it is defined with private visibility mode.

In This stmt s.sid is accessing by Non-Memberfunction(Main) it raised an error.

So for this reason data abstraction come into existence.

So to pass the data we need a public Member-function, that means

Class contains data members & Member functions. If is called encapsulation.

Test 2:

```
class student
{
  int sid = 678; // Invalid.
  char sname[20];
  float fees;
};
void main()
{ clscr();
  student s;
  getch();
}
```

→ ERROR, Since a Class is a logical representation it doesn't have any physical representation, We can't initialise any data members in the Class, this is the same rule can be applyed for Structures concepts also (C, C++)

Test 3:

```
class student
{
  int sid;
  char sname[20];
  float fees;
public: int M1;
};
void main()
{
  clrscr();
  student s; // M1 = 45; // Error.
  s.M1 = 45; // Valid.
  getch();
}
```

→ Any class properties (or) member functions are accessable only w.r.t object.

29 Test-4:

```
Struct Student
{
int
private: Int sid;
         Char Sname[20];
         float fees;
public:  void Get-data() //callee
         {
           - - -
           {
           }
         void putdata() //callee
         {
           - - - -
           }
}:
void main()
{
  - - - -
}
```

→ Every class program of c++ can also be executed by the structures concept.

- the structures of C++ contains Data members & member functions
- the only differences the class & structure in C++

Class by default is private

Structure is by default public

*Note:

Empty size class is 1 byte. (Compile will define it)

```
Class Student
{

};
void main()
{
  Class C1;
  Student S;
  Cout<<sizeof(S);
}
```

Private: Whatever the data we defining private visibility mode they can be accesable only with in the class. 30

They are visible outside a class.

a Nonmemberfunction can't access private members of the class. (Main)

It is only m.functions which can access private members of the class.

The Advantage of the private scope is data hiding Values stored in the Object are hiden from Non-member function they can be accessed only within the class member functions.

→ Data hiding is an important concept of oops.

→ Private data is hiden outside a class, but not public data.

→ A Member function can access private members directly but not non member functions.

→ a N. member function must call m. functions for accessing private members. i.e Indirect accessing.

→ Declare Variables under private scope. since Variable Contain data & we must hide the data.

→ If functions are not declared under private scope. they can't been called from the N. member function.

→ An Object becomes ~~the~~ blackbox when the Controle is in Nonmemberfunction, thus a N. member function can never access object contents, it must call a member inorder to access the contents of the Object

• Public: public Members are visible throughout the program (inside & out-side).

→ There is no data hiding for public members.

→ Both Member & Non-member functions can access public members of the class.

→ If Variables are public, data hiding is lost. i.e Values Stored in the object are not hidden from non members functions. hence avoid declaring public variables. however functions Can be public.

→ Functions Contains Struct's, we did't hide the functions.

→ Use private Variables & public functions in the class.

→ Private & Public members can appear in any order.

There is No differences b/w Structures & class in C++. however the default Scope of class is private, Structure is public.

Defining the Member functions outside the class definition:

When we define the Member function inside the class-definition by default they act as an inline functions.

We can also define the m. functions definitions outside the Class, when we define outside the class definitions, we have to follow Some rules & Regulations.

1. We have to define the function prototype inside the class.

2 By default the definition of a function will not act's as inline function, we have to provide explicitly the inline keyword.

3. The M.function definition will be accessed by the classname, with the help of scope access operator. (::)

Syntax:

```
inline Returen-type Classname :: mem-fun(arg1, arg2 ... argn)
{
  ---------
}
```

→ write a program accept any class, provide the definitions outside the class.

```
Class person          → Save this file as '.h'  'Vission.h' - filename.
{
  Char name[20];
  int age;
  float height;
public:
  Void accept();
  Void display();
};
```

Open a new file provide the definitions,

```
#Include "c:\Vission.h"
inline Void person :: accept()
{
  Cout << " Enter the Name of the person: ";
  Cin >> pname;
  Cout << " Enter age: ";
  Cin >> age;
  Cout << " Enter height: ";
  Cin >> height;
}
inline Void person :: display()
{
  Cout << " person Name is: ";
  Cout << pname << endl;
  Cout << "age: ";
  Cout << age << endl;
  Cout << " person Height: ";
  Cout << height << endl;
}
```

23

Save the file with .h extension as "controle.h"

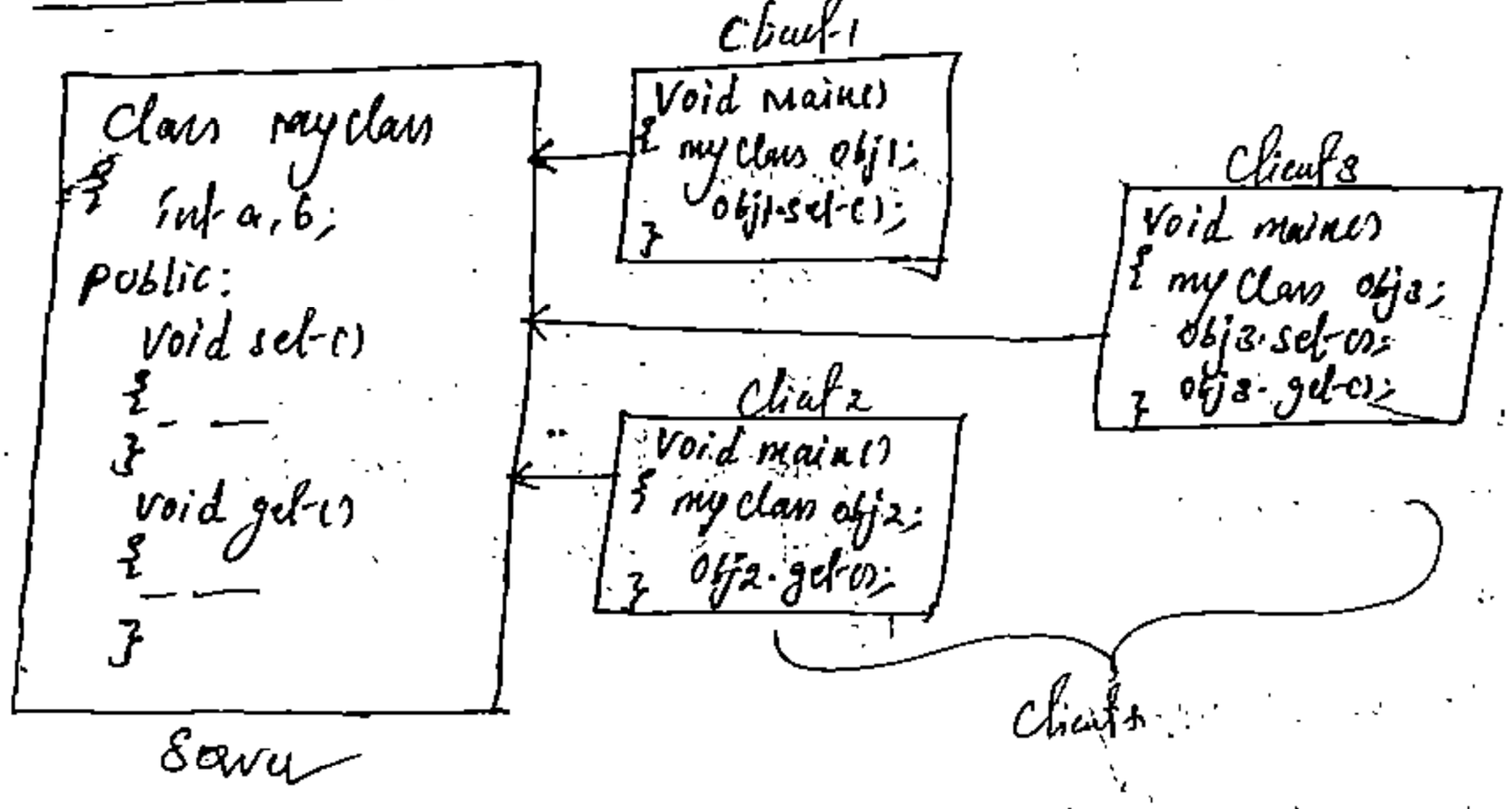
Open a new file provide the execution file.

```
#include <iostream.h>
#include <conio.h>
#include "c:\controle.h"
void main()
{
  clrscr();
  Person P1, P2;
  P1.accept();
  P2.accept();
cout<<" P1 details...\n";
  P1.display();
cout<<" P2 details ...\n";
  P2.display();
  getch();
}
```

Save the file Vicw-Cpp

** Advantages & Disadvantages of placing inside the class, outside class member function definitions.

Inside the class:



→ Suppose — client 1 requesting the server at 11:45 pm, so whenever the client 1 get's connected to server, then this server is locked until the request is proceding, so no other processes are allowed b/w that time.

→ Suppose at the same time 12:47 pm the client 2, then it is not allowed to accss the server. even through the client 2 is requesting another method (or) function which is not use by the client 1 at the same time.

→ <u>So the time comsuming & process not speed</u>. hence the queues are used in realtime datastructure. Enable to place in waiting proces. so Client 2 & Client 3 are kept in queue until the processing of client 1.

→ Suppose if one project having 1500 definitions inside the class then one method is executing, then other 1499 definition are locked, unnecessarily placed in queue.

→ <u>We can place the method definitions inside the class in which only one client at a time is allowed.</u> such type of projects.

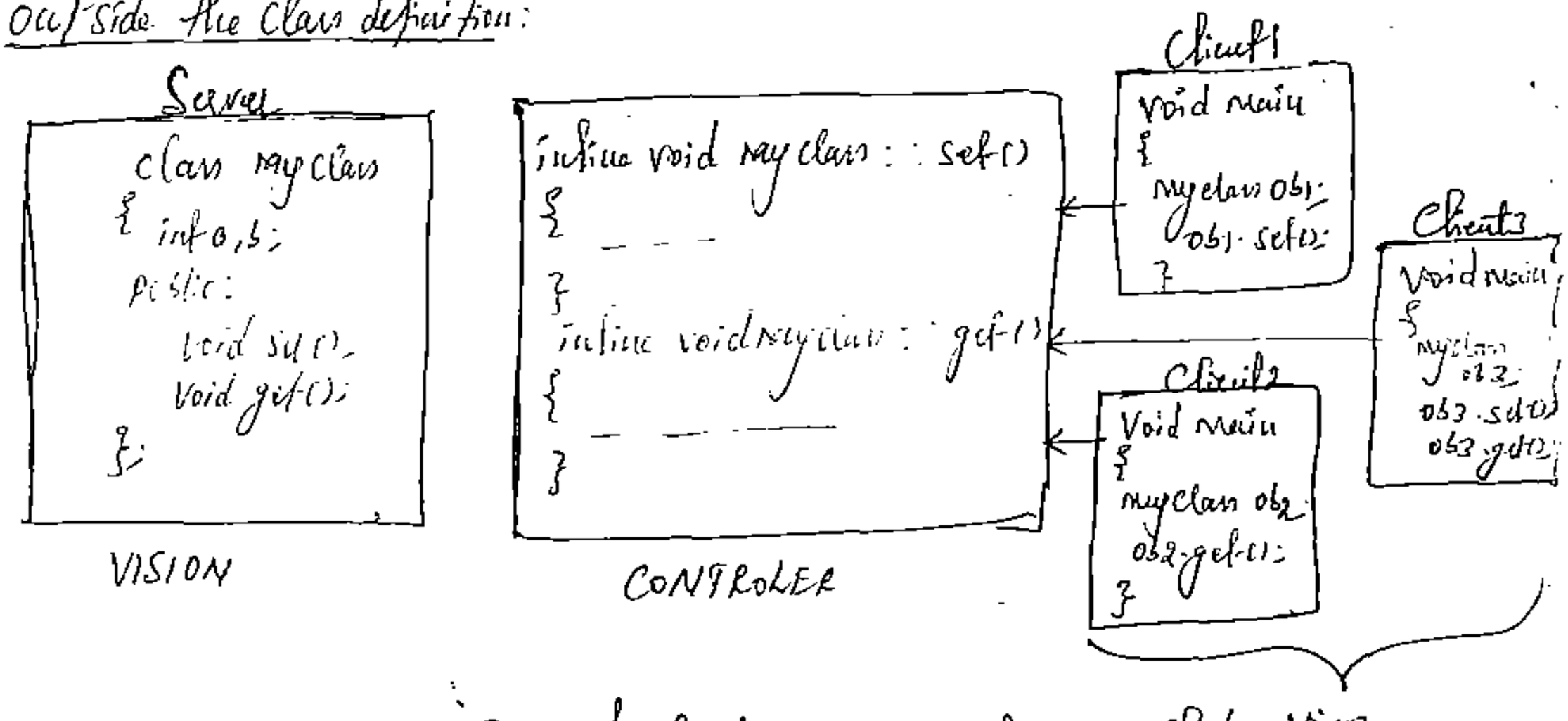
<u>EX</u>: Railay reservation online

→ In reservation PNR no is allowed differently for each client (or) passinger. so at a time only one client is allowed to accss the class.

→ So we have to plan the class properly accoeding to our requirements, were the definitions out-side (or) inside.

→ 98% of all class definitions are available outside the class.

Out side the class definition:



→ Suppose the 3rd clients comes with 2 method definitions, then only such client is put in queue until both previous 2 clients are ~~released~~ realesed the lock

→ The Advantage is maintenence, if any problem comes to the definition, then entire class should not beed repaired. So placing out-side the class definitions is good for the networking clients.

→ Speed increases as each client executing separatly, so the I/O productivity ~~wont~~ fastly will tasks place

⇒ Create a class of chair

```
Class chair
{
    char ch[20];
    int l;
    char c[20];
public:
    void accept();
    void display();
};
inline void chair :: accept()
{
    cout<<" Enter name of chair:";
    cin>> chname;
    cout<<" Enter no.of legs :";
    cin>> l;
} inline void chair :: display()
{
    cout<<" Name of chair:";
    cout<<"chname;
    cout<<" No of legs:";
    cout<< l;
    cout<<" Colour of chair:";
    cout<< c;
```

```
Void main()
{
  class2();
Chair ch;
ch.accept();
ch.display();
getch();
}
```

Silabus With parameter passing

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

Class Chair
{
  ~~Char~~
  int legs;
  float cost;
  Char *color[20];
Public:
      void set(int, float, char*);
      void get();
};
inline void chair::set(int i, float c, char *t)
{
    legs=i;
    cost=c;
    color=t;
}
inline void chair::get()
{
cout<<" No.of legs :"<<legs<<endl;
cout<<" cost :"<<cost<<endl;
cout<<" color :"<<color<<endl;
}
```

Use Enumerations to Minimise the memory using Structure Union.

36

```
Void main()
{
  clrscr();
Chair Nilkamal;
~~Chair~~ ~~Nilkamal~~.
Char Choice;
do
{
  Nilkamal.set(4,400.00,"White");
  Nilkamal.get();
cout<<"Do U want to Continue (Y/N)";
cin>>Choice;
}
while(Choice=='Y'||Choice=='y');
}
```

Here pointer is using Char it takes no.of characters.

40

→ Create a class called as ~~train~~ triangle.

37

```
#include <iostream.h>
#include <conio.h> <math.h>
class triangle
{
  private: float a, b, c;
  public:
        void getdata();
        void test();
        float area();
        float peri();
};
inline void triangle::getdata()
{
  cout<<"Enter the values of (a, b, c) 3 sides triangle:";
  cin>>a>>b>>c;
}
Void triangle::test()
{
  if((a+b>c) && (b+c>a) && (c+a>b))
    {
  else
    {
     cout<<"Not an triangle \n":";
      getch();
      exit(0);
    }
  }
inline float triangle::area()
{ float s;
  s = (a+b+c)/2;
  Return (sqrt(s*(s-a)*(s-b)*(s-c)));
}
inline float triangle::peri()
{
    return (a+b+c);
}
```

```
Void main()
{
  clrscr();
  triangle t;
  float ans1, ans2;
  t.getdata();
  t.test();
  getch();
  ans1 = t.area(); returns float
  ans2 = t.peri();
  cout<<" Area of triangle:"
      <<ans1<<
  <<"perimeter of tangle:"<<ans2
                    <<endl;
   getche();
}
```

return *this;

inline ostream & ~~class~~ ostream::operator<<(unsigned char *
return *this))

'this' pointer:

```
class Test
{
  int a,b;
 public:
        void get()
        {
          a=10;
          b=20;
        }
};

        void main()
{
   Test obj1;
   Test obj2;
  obj1.get();
  obj2.get();
}
```

Note:

→ The obj1 object invoking the member function & the member function we are assigning values for the data member. for 'a & b'.

This values indirectly defined to the obj1. i.e whichever the member function ~~invoking the~~ invoked by object.

→ This is happening by the concept of pointers, similarly like

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

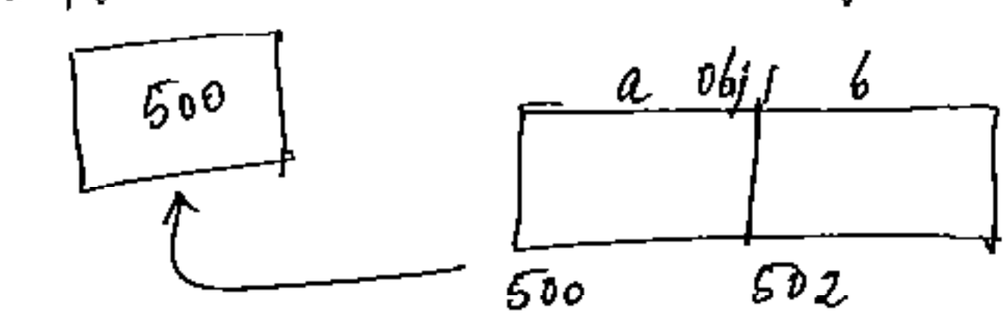
o/p. of a = 10.

→ We not define any type of pointer explicitly, but internally the object data is effecting.

Since every member functions contains an implicit pointer. that pointer acting as call-by-address concept.

By the call-by-address if you defining the values for the data members, indirectly it is effecting to the object. Since object address address will read by this pointer.

this = &obj1;



(*this).a = 10
(*this).b = 20;

internally it treates (*this)->a

→ this pointer holds the address of the objects, which invoks the function.

→ Each class member function contains an implicit pointer of it's class type, named "this"

→ The this pointer automatically created by the Compiler contains the address of the object, through which the function is invoked.

→ The this value is a constant pointer. You can't assign a value for this.

```
Void get()
{
  Test obj; // valid.
  this = &obj; // invalid. we can't assign.
}
```

→ The this pointer contains the starting address of the class data member. & it points to that particular address.

→ The this pointer acts as an implicit argument for all the member function.

→ The this pointer automatically passed to member function when it is called this is this because the computer memory of the address can't been duplicated.

→ The this pointer is used implicitly when overloading the operators using member functions.

for clarification of above stmt watch the "iostream.h"

→ The important application of the this pointer, is to return the object it points to.

→ The this pointer will be used in such cases when Two (or) more objects inside a member function should be compare & should return the invoking object as a result.

→ Using this pointer any memberfunction can find out the address of the object, to which it is a member.

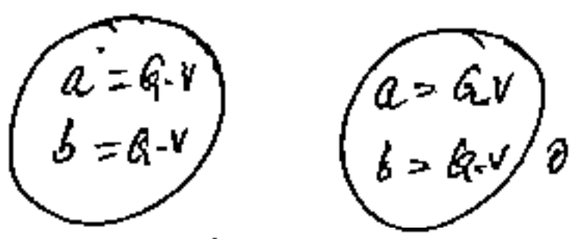
~~Start~~ Defining with this pointer any method (or) member function it will helpful to increasing the clarity of the programming

Notes

When ever we create an object, the size of an object is created with the equal size of Datamembers (variable)

Suppose 2 objects are created for 2 objects ~~same~~ memory based on it Datamembers are created. and it's ~~defen~~ names stored on it.

int a,b;



```
#Include<iostream.h>
          <Conio.h>
Class MyClass
{ Int a,b;
  public:
        Void get(Int x, Inty)
        {
          (*this).a = x;
           b = y;
        }
        Void put()
        {
cout<<"Address of this:"<<this<<endl;
 cout<<"A:"<<(*this).a<<endl;
 cout<<"B:"<<this->b<<endl;
        }
};
    Void main()
  {
     Clrscr();
    myClass obj;
  Cout<<"Address of object 1:"<<&obj1<<endl;
   obj1.get(10,20);
   obj1.put();
  getch();
   myClass ob2;
 Cout<<"Address of object 2:"<<&obj2<<endl;
  obj2.get(30,40);
  obj2.put();
  getch();
}
```

o/p: Address of object 1: 0 0x8fcdfff2
Address of this: 0x8fcdfff2
A: 10
B: 20
Address of object 2: 0x8fcdffee
Address of this: 0x8fcdffee
A: 30
B: 40

→ The internal concept of this pointer is totally implimented by call-by-address. Since the pointer ~~Simple~~ (*) can be place ~~Output~~ x not for reference concept

Only for address concept

✓→ If it is a reference Concept it is not possible every time to change address, since if is a constant

→ Every time the this pointer is holding diff types of addresses.

Applications of "this" pointer: (Where we can use this pointer)

→ Create a class as ~~measurements~~ called as measurements with the data ~~represents~~ members feet and inches, accept the Objects implement the addition operation of feet and inches.

```
#include<iostream.h>
#include<conio.h>
Class Mesu
{
  private:
      int feet,inch;
  void adjust()
  {
    feet = feet + (inch/12);
    inch = inch%12;
  }
public:
      void read()
      {
       cout<<" Enter the feet:";
       cin>>feet;
       cout<<" Enter the inches:";
       cin>>inch;
      }
```

```
Void Wife()
{
  cout<<" Feet:"<<feet<<" "<<"inches:"<<inch<<endl;
}
Mesu Sum(Mesu m)
{
 /* Mesu t;
  t.feet = feet + m.feet;     logic 1
  t.inch = inch + m.inch;
  return t; */
  feet = (*this).feet + m.feet;    logic 2.
  (*this).inch = inch + m.inch;    // This logic is proper
  return *this;                    way of utilising 'this' pointer.
 }
};
```

→ 'this' pointer contains the data which contains member function.
If more than one member function based on it declare properly.

```
Void main()
{
 clrscr();
 Mesu m1, m2, m3;
cout<<" calling m1 object---- \n");
 m1.read();
 m1.write();
 getch();
cout<<" calling m2 object---- \n");
 m2.read();
 m2.write();
 getch();
 //m3 = m+m2; invalid
 //m3 = Sum(m1, m2); valid.
  m3 = m1.Sum(m2);
  m3.write();
  getch();
}
```

best → return this; it returns address. we want value. So use *this.

$m_3 = m_1.Sum(m_2)$

(Implicitly) → this pointer

(Explicitly) → 'm' parameter

→ m_1 data will received by this pointer

→ m_2 data received by 'm' parameter. in Sum member function

→ The stmt- $\underline{m_3 = m_1.sum(m_2);}$

The 'm' invoking the member function and passing the data implicitly, this data is receving by the *this pointer.

→ Esther we writen the programming with this pointer (02) without the compiler internally will be translated to the this pointer only.

→ when we implemented without temparary object, the total data is updating within the m1 object only.

→ The return *this; is returning m1 data only.

→ Create an application called as bank.

```
#include <iostream.h>
#include <conio.h>
Class Bank
{
 private: int Account;
          Char name[20];
          float Minbal;
 public:
          Void Openacct();
          Void print();
          void Deposit(float);
          Vord Transault-(Bank &, float);
 };
   Void main()
  { clscr();
    Bank b1, b2;
 Cout-<<" open the Account for b1... in");
  b1. Opacct-();
  b1. print();
 getch();
```

45

```
cout<<"open the account for b2...\n";
b2.openacct();
b2.printb();
getch();
cout<<"Enter the amount for deposit into b1...\n";
float amt;
cin>>amt;
b1.deposit(amt);
b1.print();
getch();
cout<<"Enter the withdraw amount from b2...\n";
cin>>amt;
b2.withdraw(amt);
b2.print();
cout<<"Enter the amount to transfer from b1 to b2";
cin>>amt;
b1.Transamt(b2,amt);
b1.print();
b2.print();
getch();
}
Void Bank::openacct()
{
cout<<"Enter the Accno:";
cin>>Accno;
cout<<"Enter the name:";
cin>>name;
cout<<"Enter the minimum bal:";
cin>>minbal;
}
Void Bank::print()
{
cout<<"Accno:"<<accno;
cout<<"Name is"<<name;
cout<<"minbal:"<<minbal;
}
```

```
Void Bank :: Deposit (float, amt)
{
    minbal = minbal + ~~dant~~ amt;
}
Void  Bank :: withdraw( float amt)
  {  if ( minbal - amt >= 500)
       minbal = minbal - amt;
     else
   Coat<<" Insufficient balence .. ";
  }
Void Bank :: Transamt ( Bank &b, float amt)
 {    if( minbal - amt ) >=500)
       {  ~~bal=~~
          minbal = minbal - amt;
       // (*this) . minbal =(*this) . minbal-amt;
          b . minbal = b . minbal + amt;
       }
      else
    Cout<< " out of balence --- ";
 }
```

→ The ~~sfo~~ Afc no automatically has to generates. (Static data member).

⟷ The objects has to invokes by themself & by destroyed themself.

Constructors and Destructors:

According to oops theory every class should have one special kind of method. i-e always allocated to get initialised as soon as the object of ~~the~~ class is created.

→ Constructors give dynamic behaviour to object. (self intelegence Nature)

→ The class has to define with constructor as per the oop's theory.

→ The Constructor Name & class Name will be same in C++.

→ A Constructor will be defined is public scope. if you define is private, it's self initialisation Nature willbe lost

* doubt

→ constructor doesn't accept any return type. even void also.

→ We can't trace it out the address of the constructor.

→ Constructor's can accept the args, can be over loaded.

→ Constructors can never participate in Inheritance

→ They doesn't support for virtual Constructors.

→ Basic idea of constructor. whenever an object is is moving from

→ Method never give any dynamic behaviour, but only Constructor give dynamic behaviour.

→ Method can't have the self intelligence Nature which of the object is created by class.

→ Whenever an object is initialised it is always to adopt the boundary state and the required behaviour has to define to that particular boundary.

Constructors are of three types.

1. Default constructor
2. Parameterised Constructor.
3. Copy Constructor.

1. default:

1. Write a program to create by defining dynamic behaviour

```
#include <iostream.h>
#include <conio.h>
 Class Test
 {
  private:
          int a;
  public:
          Test()
          {
            a=10;
          }
          void print()
          {
          cout<<"A)"<<a<<endl;
          }
};
  void main()
{
    clrscr();
cout<<"Calling with Normal objects..\n";
  Test obj1;
  obj1.print();
  getch();
cout<<"Calling with Normal Object... \n";
  test obj2;
  obj2.print();
  getch();
cout<<"calling reference type..\n:";
  test obj = obj1;
  obj.print();
  getch();
cout<<"calling pointer type...\n:";
  test *ptr;
  ptr = &obj2;
  ptr -> print();
  getch();
cout<<"Calling Dynamic object..\n
  ptr = new test();
  ptr -> Print();
  getch();
};
```

→ Normal object and dynamic object involves the constructor, the basic advantage of default constructor is to decrease the no. of definitions with in the class criteria.

→ Default constructor used to in objects, their behaviour is for an environment whichever changes.

→ For default constructor we can apply default args. and here the values what we define for the default args the same values can apply for the constructor.

→ <u>Defining default args for the default constructor is always best.</u>

→ Create a class with default constructor by applying default args.

```
# Include < iostream.h>
# Include < conio.h>
#Include < proces.h>
 Class Bank-SB
{ private : int ac-no;
            char name[15];
            float amt;
  public:
        Bank-SB (int AC=1000);
        Void accept();
        Void show();
};
Bank SB:: Bank-SB(int AC)
{   if( AC<500)
    {
```

```
    cout<<"\n Minimum balance should be 500!";
        exit(1);
    }
    else
    {
      amt=Ac;
      accept();
    }
}void BankSB::accept()
{ cout<<"\n Enter the Account no:";
  cin>>acc_no;
cout<<"\n amt tobe deposited:"<<amt;
    return;
}
  Void BankSB::show()
  {
   cout<<"\n account no:"<<accno;
    cout<<"\n ~~account~~
              amount deposited:"<<amt;
  cout<<"\n Name of person:"<<name;
     return;
}
 Void main()
{
  int amount;
  char replay='y';
  do
 {
Cout<<"Enter the Amount tobe deposited:";
Cin>> amount;
 BankSB objSB(amount);
  objSB.show();
Cout<<"\n Wanna Continue[y/N]?";
 Cin>> replay;
```

```
while (replay == 'y' || replay == 'y');
cout << "\n Calling without entering the amt.";
Bank_SB obj1; // calling with
obj1.show();
}
```

2. Parameterised constructor

We implement the parameterised constructor to overcome the draw back of default constructor.

If we want to change the boundary according to our requirement not from initialisation than we use parameterised constructor.

→ If you want to creates the objects with some desired values then use parameterised constructor.

Ex:- Gaming logics are implemented by parameterised constructor. Alarm in cell phones etc are designed by parameterised constructors.

→ Write a program by satisfying a parameterised constructor.

```
#include<iostream.h>
#include<conio.h>
Class point
{
   int x,y;
   public:
      point(int a, int b)
      {
        x = (a>75)?25:a;
        y = (b>80)?80:b;
      }
   void show()
   {
     gotoxy(y,x);
   cout<<"(x,y): c"<<"("<<x<<", "<<y<<")";
   }
};
   void main()
{
    clrscr();
   point a(10,10);
    a.show();
   char replay = 'y';
    do
{ int r, c;
cout<<"\n enter new position (r,c):";
    cin>>r>>c;
    point b(r,c);
     b.show();
Cout<<"\n Wanna Continue [y/n]:";
   Cin>>replay;
}
```

```
}while(replay=='y'||replay=='Y');
```

(x,y) : c(1,1)
Wanna Continue [y/N] =

o/p: (x,y) : c(10,10)
enter the new position: 1,1

goto
1st row 1st column.

Destructors:

When the scope of the object is finished, immidiatly destroy the object by itself. then such behaviour will be takes place only by the destructors.

Destructor names & the class name will be the same name.

Destructors also be define within the public scope.
destructors willbe takes place with the simbole called as ~~it~~ "tidle" (~) followed by class name

destructors also doesn't accept any returntype even a void also.

Destructors doesn't accept any type of arguments.

Destructors doesn't contains any functionalities.

For any no. of constructors, only one destructor will be takesplace

→ Destructors supports virtual destructors.

Constructor ~~follow~~ .

→ Destructors follows bottom to top approch.

→ Write a program on destructors.

```
#include<iostream.h>
        <conio.h>
Class Sample
{
  public:
      Sample()
      {
        cout<<"Constructor invoked:"<<++i<<endl;
```

```
    } ~Sample()
      {
    cout<<"Destructor is invoked:"<<i--<<endl;
      }
};

void main()
{
  Class();
     Sample S1,S2;
       {
    Sample S3,S4;
       {
        Sample S5,S6;
       }
getch();}
```

→ Constructor invoked Top to bottom.
→ Destructor invoked Bottom to Top.

In this prog 6 objects are created by the constructor from Top S1 to S6.
Destroyed from S6 to S1.

→ Constructor & Destructor is works based on scope you specify.

ex: { Sample S7,S8;
}

→ In the above prog the constructor following Top to bottom approch. destructor buttom to top approch based on the scope.

→ As 'i' value to clearly define one is following top to bottom, other is following bottom to top approch.

→ According to the theory of Object-oriented designing & object-oriented analysis. Our object is never created unfile constructor is created

So if the Constructor is not there in our class then the object is not created, but when at the time of compilation the compiler checks, wether are created class & constructor (or) Not.

If the constructor is not there, then as our Compiler is an Object-Oriented Compiler it is implicitly creates a default constructor, so C++ is called an Object-Oriented programming.

So even if you do not design your class as object Oriented, C++ compiler is converted into object oriented by placing

default constructors.

Not only default constructors the C++ compiler also support overloaded assignment operator & copy constructor, by default.

ex:1

```
Class ex
{
};
Void main()
{
  ex e1;
  ex e2; // 0-Argument constructor
  e2=e1; // Overloaded assignment operator
  ex e3=e2; //Copy constructor.
}
```

// Compiler provide only default constructor only @ When you not call any constructor.
If by default ~~only~~ C++ can't takes parameterised constructor

~~Not~~ How you prove your C++ Compiler is a OO compiler?
All ex:1, ex:2.

ex:2

```
Class ex
{ int a;
public: ex(int x)
        {
          a=x;
        }
};
Void main()
{ ex e(0);
  ex e1;
}
```

⟶ Error. Since we explicitly provided parameterised constructor, so explicitly also provide default constructor, at this time Compiler will provide.

either the user has to provide all the constructors (or) Compiler will provide.

O/P: Error.

Nameless object:

```
// Nameless object
#include <iostream.h>
#include <conio.h>
Class nameless
{
```

```
Private: int a;
public:
        namelen ()
        {
          a=10;
          print ();
        }
        void print ()
        {
        cout<<"A:"<<a<<endl;
        }
        ~namelen ()
        {
        }
};      }
void main ()
{ clrscr();
  namelen ();
  getch ();
}
```

→ The Advantage of nemden objul- immediate destructions willbe takesplase

→ providing a parameterised constructor indirectly youre are providing type conversions from basic C predefined) to class type Cuser defined.

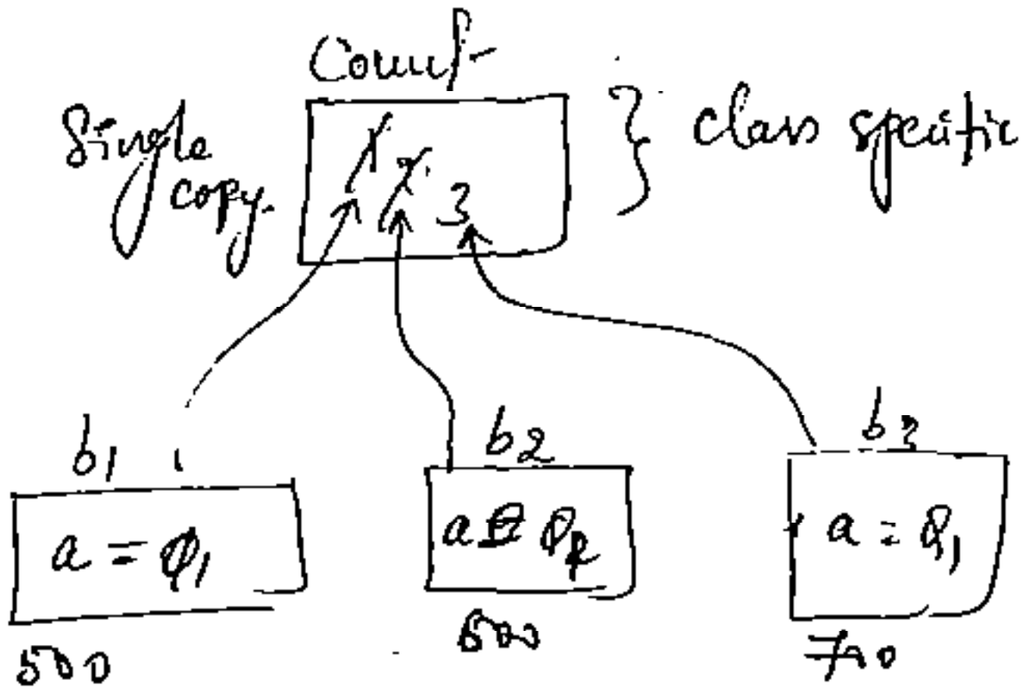
Static Data members & Member functions.

```
Class Bank
{ int count;
public:
        Bank ()
        {
          count=0;
          count++;
        }
};
Bank b1, b2, b3;
```



```
Class myclas
{ private:
        int a;
  public:
        ---
        ---
};
Void main ()
{ myclas obj1, obj2, obj3;
```

```
Class Bank
{
  int a;
Static int cocunt;
 Void Bank;
{
  cocunt ++;
   a=0;
  a++;
}
};
  Bank b1, b2, b3;
```



→ In class we can define data members and member functions of a class can be Static

→ When you preceed a member variable declaration with a keyword of Static. you are telling the compiler that only one copy of one variable will exist. and all the objects of the class will share that.

→ Unlike regular datamembers individual copies of a static member variable are not made for each object no matter how many objects of a class are created. only one copy of the static datamember is exists therefore Static datamembers can be set to be class specific not instance specific.

→ It's existance is to the class not to the object of the class.

→ When you declare a static datamember you are not defining it (you are not allocating the memory)

→ To allocate the memory you must provide a global definition for it. else where outside the class. This is done by redeclaring a static variable using

Scope resolution operator (::)

→ This statement the storage for the variable to be allocated.

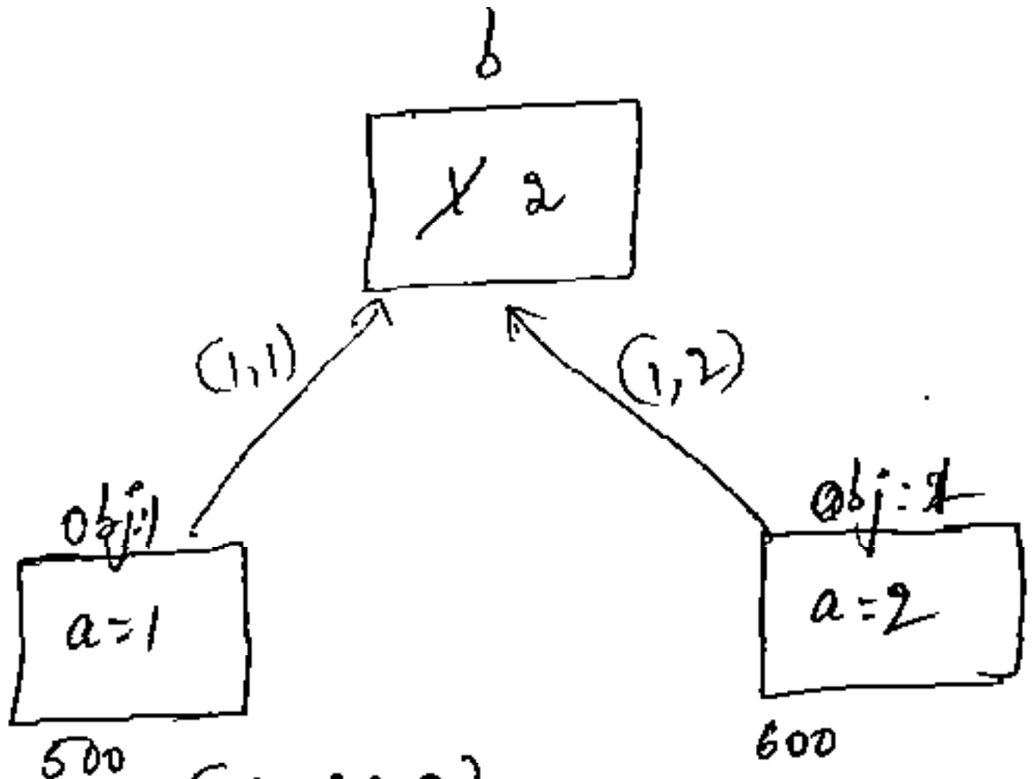
→ The static member variable is existed if before any objects of it's class is created

→ A static data member in both private and public can be existed.

→ Since a static data member exists before an objects of all the class is created. it can be given a value at any time. Therefore the value of a static variable is unchanged by the creation of the object.

```
#include <iostream.h>
#include <conio.h>
class static_demo
{ private:
        int a;
    static ~~static~~ int b;
  public:
      void set (int i, int j)
      {
        a = i;
        b = j;
      }
      void print()
  };
      inline void static_demo :: print()
      {
       cout << " Non static a:" << a << " static b:" << b << endl;
      }
        int static_demo :: b; // definit
```

```
void main()
{
 clrscr();
 static_demo obj1;
  obj1.set(1,1);
  obj1.print();
  getch();
  static_demo obj2;
  obj2.set(2,2);
  obj2.print();
  getch();
  obj.print();
  getch();
  cout<<"sizeof(obj1);
}
```



Ex: 2:

```
class counter
{ public:
    static int count;
    counter()
    { count++;
    }
  ~counter()
  {
    count--;
  }
};
```

```
int counter :: count=0; // define
  void t(); // declaration
void main()
{ clrscr();
 counter c1;
cout<<"object in existence" <<
    counter :: count<<endl;
  getch();
  counter();..
cout<<"object in existance"<<counter::
                  count<<endl;
  t(); // caller
cout<<" object existance :"<<counter::
  getch();          count<<endl;
}
```

```
void t()
{ counter temp;
  cout<<"object in existance"<<counter :: count<<endl;
```

→ At last all the objects are ~~destor~~ destroyed.

→ In this program the count data members is defined as static and it is define in public so we can acess outside the class by the name of class with the help of scope resolution operator.

→ Counter :: count ;// Valid (Counter is class name).

→ c1.count ;// Invalid (Since count is an class specific)

→ In this program if you find the size of the object it is only 1 byte

Static Member functions:

Member functions can also be declared as static. Static member functions have the restrictions this may directly refuer to static members of the class. i.e static member function can be initialise only static data members they can't initialise non-static datamembers.

→ The static member functions owe good usage Initialise Private static data before any object is created.

```
class stat_demo
{
  private:
    static int a;
  public:
    static void set(int i)
    { a=i;
    }
```

```
    void printf()
    {
      cout<<a;
    }
};
int Stat-demo::a;// definit
  void main()
 {  clrscr();
 Stat-demo::set(100);// Initialisation of Static datamember
                        before object creation.
Stat-demo s;
   s.printf();
getch();
}
```

→ This pointer will not works in Static member function Since there is no need of invoke Static member function by the object. i.e object is not existed means this pointer will not exist.

→ We can't apply Static for Constructors.

→ Many applications are defining under Static and this application will not exist without Static

1. Inheritence never exist without Static.

2. Java never exist without Static.

Defining Static it give global accessing, no need to create Separate instance for each class.

public Static void main(String [] args)

→ A Unix O.S no two processor id's will not be the same.

→ No two Clients will not have the same railway reservation PNR-Number

Singleton class: only one object will be created from a class is called singleton class, some classes has to create only single object such logic of classes can be designed then they are called as singleton classes.

→ When we want to design a class, it doesn't possible to create more than one object we have to take some classes which required single object.

for ex:- A Computer pheripherals which is allocating a singleton classes like one monitor, keyboard, etc. if design this classes we have only single object.

→ To implement singleton classes, we required a static member function & private constructor.

```
class Sample
{
private:
    int i;
static Sample *p;
    Sample()
{   i=9;
}
public:
  Static Sample *Create()
  { if(P == NULL)
    P = new Sample;
    return p;
  }
};
Sample *S = Sample::Create();
void main()
{
Sample *S1 = Sample::Create();
Sample *S2 = Sample::Create();
}
```

→ The need of private constructor we are restricted not created any objects either the static (or) dynamic from the main program.

→ The static member function can invoke from outside the class without creation of object, from the static member function inside the class we can reach to the private constructors from the class.

→ The purpose of static data members in order to access by static member function (Since static member function can not access Non static data members).

→ The data member of static is a pointer type of class since we are dynamically allocating the object memory space and invokes the constructor and that address is assigning to p, the return 'p' stmt back to the main since we called from main object & successfully created address of object is available to 'p'.

→ 'p' value is allocated by S_1, 'p' must be static since it has to return the address of the same object.

→ So S_1 & S_2 always get the same address, similarly we call from the main program any no. of times we get the same address. finally it is restricted to create one object only.

Advantages of constructors:

1. The advantage of constructor is it get the dynamic behaviour for an object.
2. All the objects have the same initialised values, how many objects are created. Suppose, if we create thousand objects we can initialise all the objects at a time.

→ This type of initialisation electronic industry will follows, initialisation with constructors is a failor only to make the object in which boundary. i.e existing, not for removing garbage values.

→ We can give perticular boundary via constructors only.

Disadvantage:

By using constructors we must give initial boundary, suppose we want to change the boundary of some objects out-of thousand objects then it is very defficult to change the boundaries for some objects due to this. It is a time consuming & decrease the speed.

Destructors Advantages:

By using destructors only intelligent can't trace out the object data. & it provides security so it is very important to destroy the object after the task is finished.

→ Static object only automatically destroy whenever the scope of the object is finished.

→ Dynamic objects must be destroy explicitly by using delete operator.

→ so destructors saving memory & provides Security

Copy Constructors:

Copying machanism is copying of one object data for another data, When copy machanism is implementing Normally compiler defines it own machanism called as default copy (or) bitwise copy.

→ All ways this type of copy machanism is not Suitable, then so user has to provide it's own copy machanism explicitly to copy one object data into another object.

→ The copy mechanism by default the compiler is supports for some cases only if is possible.

```
#include <iostream.h>
#include <conio.h>
Class bitcopy
{  int a,b;
   public:
     bitcopy(int x, int y)
     {  a=x;
        b=y;
     }
     void print()
     {
  cout<<a<<b;
};
  void main()
{
  clrscr();
  bitcopy (10,20);
  bitcopy b2=b1;
  b2.print();
  getch(); O/P: 10,20.
}
```

→ In this example one object data is copied into another object and the copy mechanism is supported by the compiler.

→ In this b1 data is initialised to b2 & the mechanism is called as bitwise copy we will not get any side effects. Since the class contains normal data members.

→ ~~In this data~~

→ When a class contains pointer type of data member then it is better to define explicit copy mechanism.

→ Once we define explicit copy mechanism then the default copy mechanism is by passed (Not present).

→ A user defined copy constructor requires when object contains it's own pointer and reference type data members such as to a file. In such cases a destructor and assignment operator should be written by the user.

→ Copying of objects is achived by a copy constructor and assignment operator.

→ A copy constructor will have it's own class name.

→ It can be present with arguments, in some of the following cases the copy constructor will ignores.

→ When one object explicitly initializes another, such as in a declaration.

→ When a copy of a objects made to be passed to a function.

→ When a temporary object is generated (most commonly as a return value of a function).

This three ways can be achieved by two techniques

1. Explicit assignment in the expression
2. Initialization.

1. Explicit assignment in the expression:

```
object a;
object b;
a=b;
```

2. Initialization: An Object can be initialized by any one of the following ways.

a). Through declaration

```
Object B = A;
```

b) Through function arguments.

```
Type function(Object a);    m3 = m1.add(m2);
```

c) Through function return value

```
Object a = function();
```

The copy constructor is used only for initialisations, doesn't applied to assignments where the assignment operator is used.

→ The implicit copy constructor of a class calls the internally & copy it's members by means of appropriate type.

→ If is a class type, copy constructor is call, if it is Scalar type. Then built in assignment operator can be used if it is array each element is copied by providing user defined copy-constructor

Example on Implicit copy Constructor (compiler provided)

```
#include <iostream.h>
class person
{ public:
    int age;
    person (int age)
    {
      this -> age = age;
    }
};
int main()
{
  person t(10);
  person t(15);
  person tc = t;
cout << t.age << " " << s.age << " " << tc.age << endl;
    t.age = 23;
cout << t.age << " " << s.age << " " << tc.age << endl;
}
```

O/P: 10 15 10
23 15 10

→ As expected 't' has been copied to the new object 'tc'. While age was changed tc, S age remained the same. This is because they are totally different objects

→ When we define the statement tc=t, the copy mechanism is provided by the compiler, the 't' data receiving by formal parameters.

→ The 'tc' data will be recev by 'this' pointer.
Which is provided by inbuilt Compiler member function.
The member function looks on this ~~pointer~~ pattern.

```
Person (const person & copy)
                          ↑ t
{
  this → age = copy.age;
   ↑ tc        ↑ t
```

→ So when we require actual copy constructor defined by the user.

User defined copy constructor:

```
#include <iostream.h>
  class Array
{ public:
     int size;
     int *data;
     Array (int size)
     {
      this → size = size;
      this → data = new int[size];
     }
     ~Array ()
    { delete [] data;
    }
};
```

```
int main ()
{ Array first(20);
  first.data[0] = 25;
  { Array copy = first;
cout << "first.data[0] << " " <<
    copy.data[0] << endl;
  } //Stmt (1)
  first.data[0] = 10; // Stmts.
}
```

O/P: 25 25

Null pointer assignment (C segmentation fault).

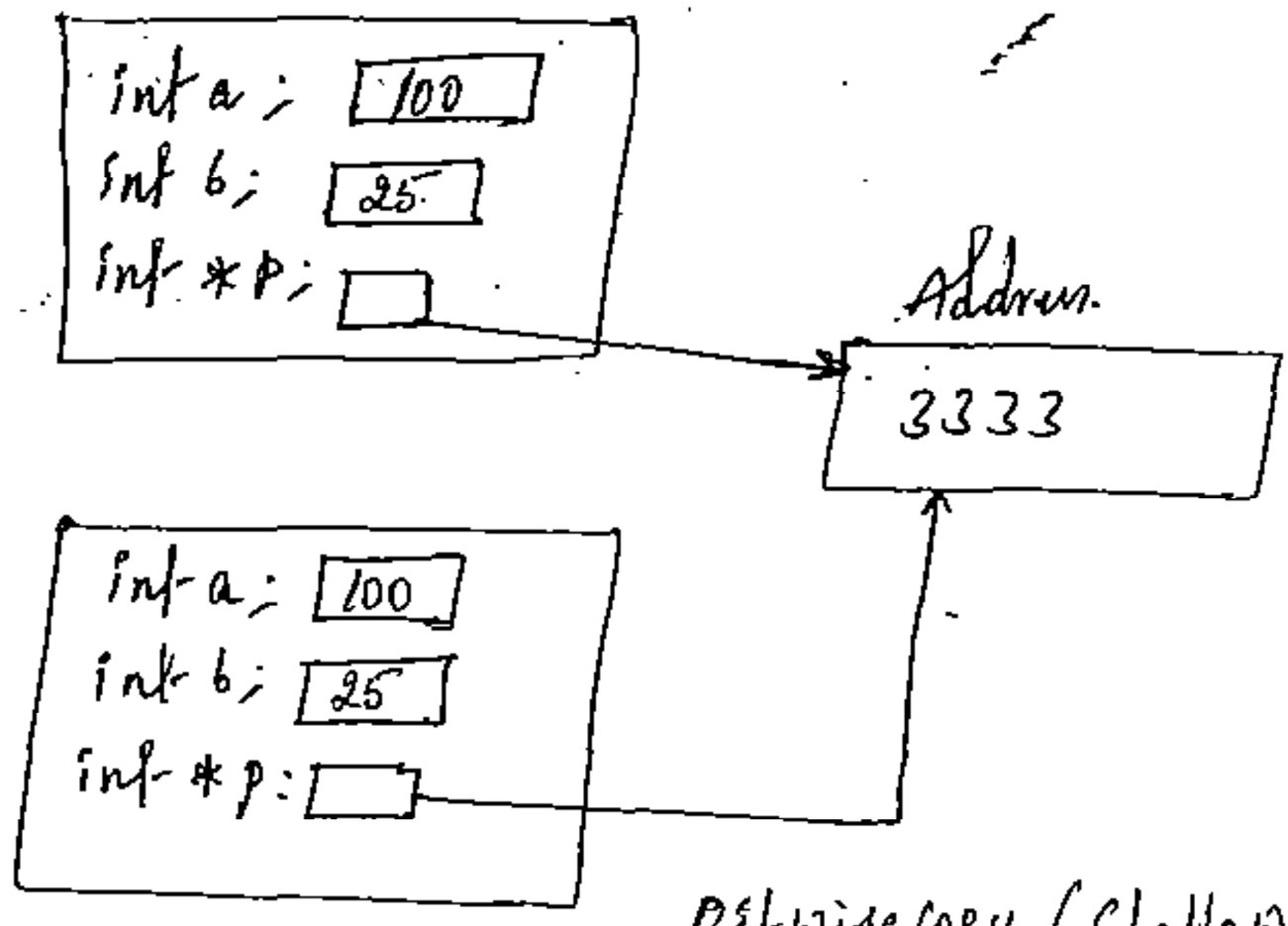
In the above program we ~~did~~ didn't specify any copy-constructor explicitly, so at this stage compiler generated a copy mechanism and execute the program.

→ The problem with this constructor is that it performs a "shallow copy" of the data. pointer it only copies the address of the original data members. this means they both share a ~~memory~~ pointer to the same chunk of memory which is not what we want.

when the program reaches line 11, copies destructor gets called (because objects on the stack are destroyed automatically when their scope ends) Array's destructor detects the data Array of the original.

Therefore when it deleted copy's data, because they share the same pointer. it is also deleted first's data line (2). now access invalid data and writes to it. this produces the Segmentation fault.

→ If we write our own copy constructor that ~~that~~ performs a 'deep copy' then this problem goes way.



Bitwise copy (Shallow copy).

→ A bit wise copy Constructor performs a simple Variable by Variable assignment copy of 'a'. The components of an object the bitwisecopy Constructor is so named as it. mainly copied bit-by-bit from the Orginal object to the new object

→ In the above example we can create a new integer array and copying the contents to it. Now copy object destructor for only deletes it's data not the first data.

→ So the line 2 will not produce any ~~errors~~ errors. this type of copy machanism provides deep copy

→ We Can provide without Segmentation fault by the deep copy but they will accept Separate memories to get without Segmentation fault.

```
Inf main()
{
  classes();
  Array first(20);
  first.data[0] = 25;
  {
    Array copy(20);
    copy.data[0] = first.data[0];
    cout << " first data[0] << " " << copy data[0] << endl;
  } // Stmt(1)
  first.data[0] = 10; // Stmt(2)
}
```

In this we define extra memory and copying is

taking place without extra memory we can optimise with less space.
We can implement by using copy constructor.

Class_name (const Class_name &) → indicates reference type
{

}

→ Write a prog implimentation of copy mechanism by explicitly defining
the copy constructor

```
#include<iostream.h>
        <conio.h>
        <string.h>
Class clsString
{
  private:
   int length lenName; // Length of String Not required for dynamic
                          memory allocation
   Char *strName; // The String required to have dynamic memory
                     allocation
  public:
   // The constructor where memory is allocated.
      clsString(char *s = " ")
      {
        lenName = strlen(s)+1;
        strName = new char[lenName];
        strcpy(strName, s);
      }
      // The copy constructor
      clsString(const clsString &t)
      {
cout<<" In copy constructor "<<endl;
  lenName = t.lenName;
  strName = new Char[lenName];
  strcpy(strName, t.strName);
```

```
// Destructor where Memory is freed
 ~clsString()
{
cout<<" In destructor Name is "<<strName<<" and length is "
 delete [] strName;                    << lenName<<endl;
}
// Returns the value where data is stored in allocated memory
 char *getName() {return strName;
                 }
// Returns the value where the data is stored Not by allocating
                 Memory
   int getLen()
  {
    return lenName;
  };
 int main()
{
   clrscr();
// Explicit Initialization
 clsString myName(" Bill Gates");
// Object 1 Data
  cout<<" Initial Object Name : "<<myName.getName()<<endl;
  cout<<" Initial Object length: "<<myName.getLen()<<endl;
// Initialization by existing object
    clsString myDuplName = myName;
      // Object 2 Data
Coat<<"Copied object Name : "<<myDuplName<<".getName()<<endl;
Coat<<" Copied Object length: "<<myDuplName.getLen()<<endl;
Coat<<"_____"<<endl;
```

```
cout << " ---------- " << endl;
return 0;
```

Note: Copy constructor always takes references, Not values (or) addresses.

O/P:

```
Initial Object Name : Bill Gates
Initial object Length 11
In copy constructor
Copied object Name : Bill Gates
Copied object length: 11
```

```
In destructor Name is Bill Gates and length
In destructor Name is Bill Gates and length.
```

1st object data

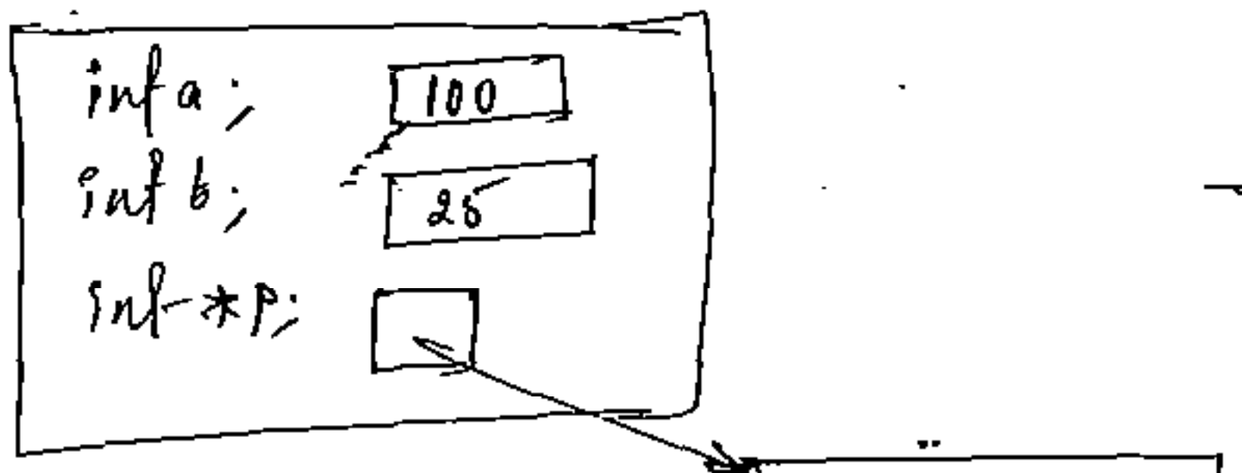
2nd object data. if is copied from 1st obj.

Destructor called for every object Obj1 & Obj2.

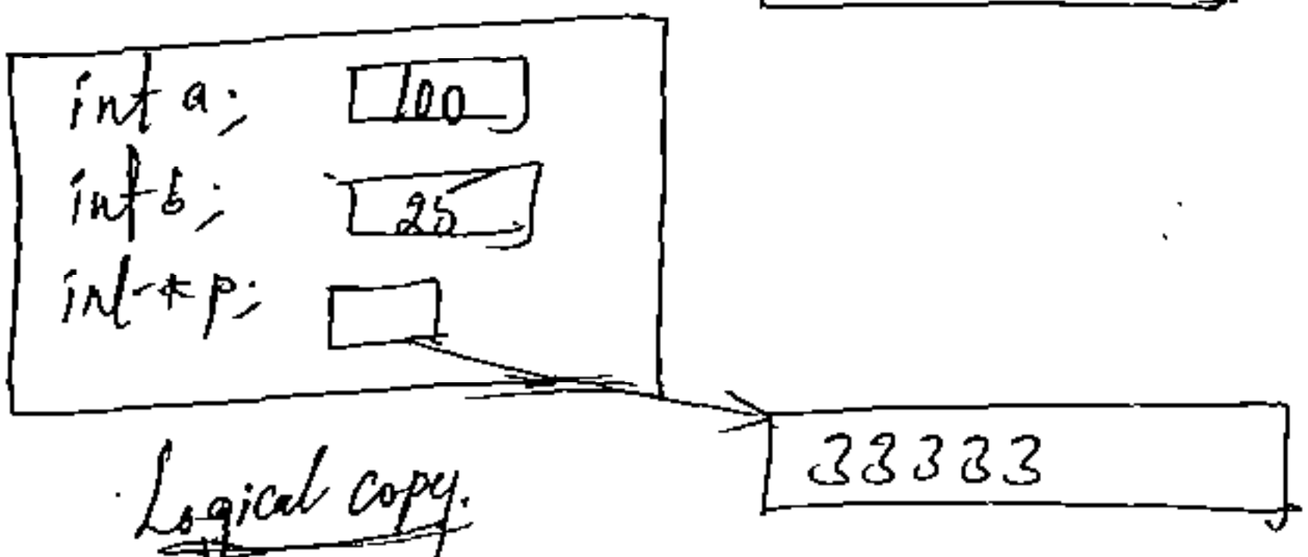
→ By the copy Constructor it is happening the logical copy.

A Logical copy Constructor makes a true copy of the Structure as well as it's dynamic Structures.

Logical copy Constructor come into picture mainly when there are pointers (or) Complex objects with in the object.



Note: Original copy & Logical copy refer has the same addrss. i.e logical copy refering the address of Original copy



Logical copy.

→ Many applications working under copy constructors.
Sending pictures, & files, from one mobile to another mobile.

→ Magnifier is working only by copy constructors.

→ ~~Aaa~~ gaming programing vastly used.

→ Remoet desktop utility.

, what i saw on the system is displayed on Screen.

Ex: Team Viewer

Array of objects with constructors:

```
Class aob
{
private:
        int a;
public:
        aob (int i)
        {
          a=i;
        }
        void print()
        {
    cout<< i <<" ";
        }
};
   void main()
{
  aob ob[3] = {1,2,3,4,5}; // Short-form of initialization
aob ob[3] ={aob(1),aob(2),aob(3)}; // Long form.
    for(int i=0; i<3; i++)
        ob[i].print();
        getch();
}
```

In this we can initialise Array of objects in Short-form as well as in Long form.

If a Constructor with 2 parameter we can't initialise in short-form, We have to initialise only in Long form.

Not only two, more than 2 parameters also possible.

8

```
Class aob
{ private: int a, b;
  public: aob(int i, int j)
          {
            a=i;
            b=j;
          }
          Void print()
          {
          cout<<i<<" "<<j<<endl;
          }
};
  Void main()
 {
  aob b[3]={aob(1,5), aob(2,6), aob(3,7)}; // Long form
      for(int i=0; i<3; i++)
        ob[i].print();
      getch();
 }
```

Inheritence

Inheritence is one of the impotent ~~core~~ corner stone for the oops concept.

→ Inheritence supports the creation of hisarical classifications.

→ When Inheritence are implementing we have to think about- Generalization & Specialization.

→ Generalization is a process that defines common set of objects with common behaviours & attributes. and generating a general class is called as <u>generalization.</u>

it will be <u>takes place</u> allways at <u>Top level</u>. it makes you to generate a Base class.

→ On the other hand taking the cmn features & generating a Special class is called as <u>Specialization.</u>

It makes you to develope a derived class. it always <u>takes place</u> at <u>bottom level:</u>

<u>def:</u> → <u>Inheritence is one type of intelligence how to extend the current software</u>, which contains some facilities what i need, & some more properties. don't have the properties which my class contains use them & design your own class with additional features.

→ Inheritence is an Object-oriented language like C++ makes the <u>data</u> & <u>methods of a base class</u> (super class) available to it's Subclass. (Derived class)

→ In keeping with C++ terminology a class that is <u>inherited</u> that is referred as <u>Base class</u>.

The class that does the <u>inheriting</u> is refered as <u>derived class</u>

→ We should extend the class without damaging the previous class logics.

→ All classes are not the base classes until & unless you extend your designed class. Unless you extend your class that doesn't became as a base class.

The present where you extend your class then only that class became as a Base class.

→ Basically a Base class designer wants to restrict the use of the class by the derived class.

→ <u>Derived classes</u> can <u>Never be cloning class</u> (same as) as the Base class. (because it has some own features).

→ A <u>Base class gives</u> <u>every thing is accessable</u> <u>by derived class</u> <u>but</u> a base class <u>never allowes to change it's properties</u>. <u>Just utilise the services of base class.</u> you can't destroy the base class features.

→ Hiding doesn't mean not accessing use in a secure way.

→ Each <u>instance</u> (object) of the derived class includes all the members of the base classes. The derived class inherites all the properties of the base class.

Therefore the derived class has largest of properties than it's base class. however the derived class will overrides some of the properties of base class.

→ Depends on the access spaitier the derivation will be takes place.

Applications of Inheritance:

1. S/w upgradation is only possible by Inheritence

Syntax:

```
class Base-classname
{
  ——
  ——
}
class Derived-classname : <visibility mode> Base-classname
{
  ————
  ——————
};
```

Base class visibility	Derived class visibility.	
	public	private
Private	Not inherited	Not inherited
protected	protected	private
public	public	private

Inherited Base class members not accessible

Ex: If the base class has been derived under the private derivation then the all the base class visibility modes can be becomes private in the derived class.

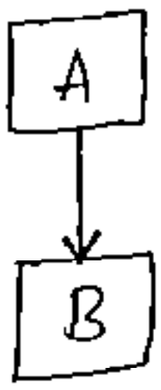
→ Basically visibility modes in the base class will be three types & derivations are of Two types.

→ The C++ supports many type of inheritence.

like 1. Single Inheritence.
2 Multilevel Inheritence.
3. Multiple Inheritence (Only C++ supports)
4. ~~Hiebar~~ Hiearchical inheritence
5. Hybrid Inheritence

1. Single Inheritence:

—The inheritence mechanism in which single derived class will be inherited from a single base class.



→ Example on Single Inheritence by public derivation.

```
#Include<iostream.h>
#Include<conio.h>
Class Base
{ private:
        int a;
  protected: int b;
  public :  int c;
     void dede()
    {
    cout<<" Enter the values: ";
    Cin>> a >> b >> c;
    }
     Void write()
    {
  cout<<" A :"<<a<<" B :"<<b<<"C :"<<c<<endl;
    }
};
```

```
Class der : public Base
{
 private: ints;
 public :
        Void sum()
         {
         S = b+c;
         }
       Void print()
    {
  cout<<"Sum:"<<S<<endl;
    }
};
```

```
Void main()
{ clrscr();
  Der d;
Cout<<" sizeof S :"<<sizeof(s)<<endl;
 d.read();
 d.write();
getch();
 d.sum();
 d.print();
getch();
}
```

Total size of object. d is 8 bytes.
base class 6 byte, derived class 2 bytes.

→ If you write a stmt in the derived class of the 'sum' member function $s = a+b+c$; if raises an error.

Since 'a' is a private property can't accessable. So we can remove the 'a' property from the base class since 'a' is unnecessary property.

Cout << d.c; If is possible. Since 'c' is public property derivation from public.

Cout << d.b; If you write this stmt in the Non-member function. the 'b' property can be accessable with in the classes not from outside the classes.

i.e The data is protected with Secure & inheritable so try to define all the data in protected when inheritence are Inkrementing.

Single inheritence by private derivation.

When we made a private derivation make the changes in read & write in the mains program.

```
#include <iostream.h>
        <conio.h>
Class Base
{ protected: int b, c;
```

```
public:
        void leadu
        {
        couf<< "Enter the values:";
        Cin>> b >> c;
        }
         Void wife-cs
        {
      Couf-<<"B:"<<b<<"C:"<< c'<<endl;
        }
};
Clan dee : private Base
{
 private: inf s;
 public:
          Voidsum()
          {
           lead()
           s= b+c;
          }
           Void print()
          {
           wlite()
          Couf<<" sum:"&<s <<endl;
          }
};

  Void mein()
{
  clascc();
   Dee d;
Coul-<<"size of d:" << sizeof(d);
 getch();
 d. sum();
 d. print();
getch();
}
```

→ Insurance

↓

payment- decivation public

```
#include<iostream.h>
#include<conio.h>
Class Insurence
{
  protected:  int no;
  public :    char name[20];
}
              void accept()
              {
```

```
Class payment
{
   private :  int n;
   public:
        void mpay()
        {
```

```
  void main()
  {
    clrscr();
```

```
  Class Insurence
{ protected: //both Inheritable & Secure
   int no;
   char name[15];
   int years;
   float amt;
  public:
        void accept_data();
        void show_data();
};
```

```
Void Insurence :: accept_data()
{
 Coul<<"\n Enter Insurence no:";
 Cout>> no;
 coul<<"\n Enter policy holders Name:";
 Cin>> name;
 Coul<<" \n Enter policy amount:";
 Cin>> amt;
 Coul<<" Enter no.of years issued for:";
 Cin>> year;
 Return;
}
 void insurence:: show_data()
 {
  Coul<<"\n Insurence No:"<< no;
  Coul<<"\n Policy holderName:"<<name;
  coul<<"\n policy amount:"<< amt;
  Coul<<"\n Issued for:"<< years;
  Return;
}
    Inheritance the base class is derived

 Class payment : public insurence
{
 private: float
          float maturity_value;
          float late_entrest;
public:  Void Calculate_ maturity()
         {
          Maturity_value = (amt * years * rate_ Sustrent.) / 100;
          Maturity_value = Maturity_value + amt;
         } Return;
  Void accept_data;
  Void display_data();
};
```

```
Void payment :: accept_data()
{ insurence :: accept_data();
 cout<<"\n Enter rate of Intrest :";
 cin >> rate_Intrest;
 Calculate_maturity();
 return;
}
 void payment :: display_data()
 {
   Show_data();
cout<<"\n maturity value after the period is";
   cout<< maturity_value;
   cout<<"\n rate of instrest is:"<< rate_Instrest;
    return;
   }
   void main()
{
   payment a;
   a.accept_date();
   a.display_date();
}
```

&item → &sales.

&no
&sales.
&salesno.

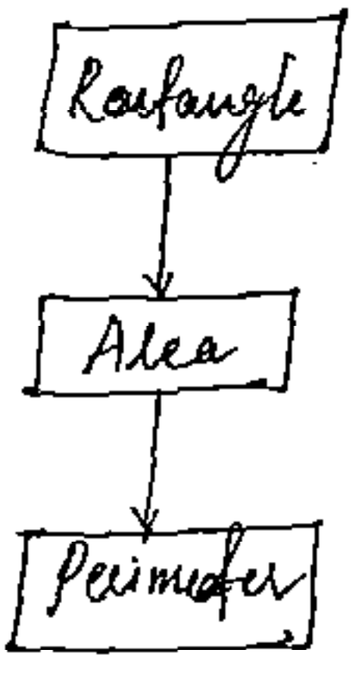
(2) Multilevel Inheritence:

The inheritence machanism in which every intermediate class will acts as an base class.

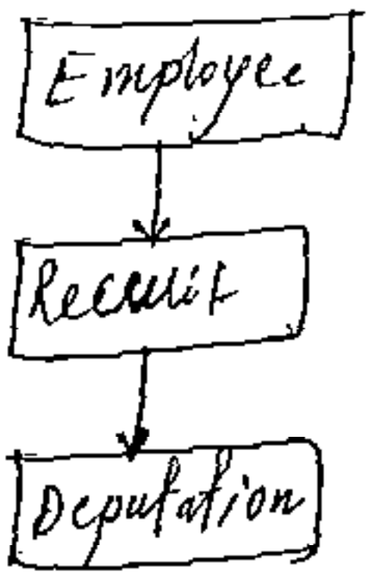
It is like Generation by Generation.

We can generate any no. of levels.

EX:



EX:

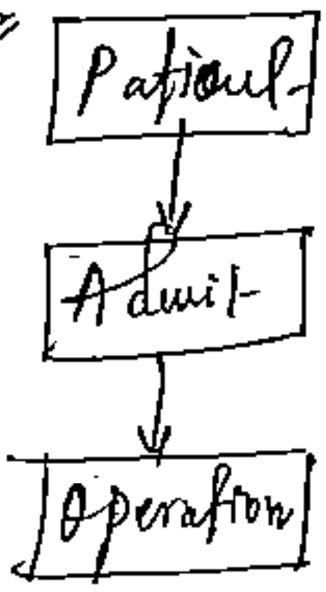



```
#include <iostream.h>
#include <conio.h>
class emp;
{ protected:
        int no, allot;
        char name[15], post[15];
  public:
        void accept();
        void show();
};
void emp :: accept()
{
  cout<<"\n Enter emp no:";
  cin>>no;
  cout<<"\n enter emp good name:";
  cin>> name;
  cout<<"\n enter post apptied:";
  cin>> post;
  cout<<"\n is the employee is allotted for any job [yes=1/no=0]:";
  cin>> allot;
  return;
}
class recruit : public emp
{
  protected:
        void emp :: show()
        {
          cout<<"\n employee no:"<<no;
          cout<<"\n employee name:"<<name;
          cout<<"\n postapplied:"<<post;
          return;
        }
        class recruit : public emp
        {
          protected:
            char dept[15], mgr[15];
              float sal;
              int depot;
          public:
              void accept();
              void show();
        };
```

```
void recruit::accept()
{
    emp::accept();
   if(allot)
  {
cout-<<"\n To which department:";
cin >> dept;
cout-<<"\n Under whoom(manager):";
cin >> mgr;
cout-<<"\n With how much salary:";
cin >> sal;
cout-<<"\n is the employee is deputed to any other company
                                    [yes=1/no=0]:";
cin >> depot;
 } return;
}
  void recruit :: show()
  {  emp::show();
     if(allot)
    {
  cout-<<"To the"<<dept<<" department::
  cout-<<" under manager):"<<mgr;
  cout-<<" with a salary:"<<sal;
     if(dept)
  cout-<<"\n the employee has been deputed:";
  }  return;
}
class deputation : public recruit
{
  protected: int deput-no;
             Char place[15];
  public:    void accept();
             void show();
}
```

```
Void deputation :: accept()
{ recruit :: accept();
  if(dept)
  { cout<<"\n Enter depatation no:";
    cin>> deput_no;
     cout<<"\n To which place deputed:";
    cin>> place;
  } return;
}
   Void deputation :: show()
  {
     recruit :: show()
     if(dept)
    {
   cout<<"\n deputation no":<<deput_no;
   cout<<"\n deputed to:"<<place;
  } return;
}
   Void main()
  {
    clrscr()
   deputation d;
    d. accept();
    d. show();
  }
```

→ In the inheritence concept we are creating the object for the derived class



Constructors, Destructors & Inheritence:

There are 2 Major questions that raise related to Constructor's & Destructors.

1. When Inheritence is involved 1st when a base class & derived class constructor & Destructor functions called?

2. How can parameters has to pass to the base class constructor functions?.

When Constructor & Destructor functions are executed if is possible for a baseclass, a derived class. (or) both to contain a Constructor, ~~&~~ destructor functions, ~~is~~ is important to understand the order in which this functions are executed.

When object of a derivedclass comes into existence & when it goes out of it's existence.

```
#include <iostream.h>
#include <conio.h>
Class Base
{
 public:
        Base()
        {
    cout<<" Base class constructor ...\n";
        }
        ~Base()
        {
  cout<<" Base class destructor ...\n";
 }; }
 Class  Der : public Base
{
```

```
Public:  Del()
         {
    cout<<" Derived class constructor...\n";
         {

         ~Del()
         {
    cout<<"Derived class destructor...\n";
         }
};
    Void main()
    {
      clscr();
      Del d;
      getch();
    }
```

O/P:

Base class constructor
Derived class constructor
Derived class destructor
Base class Destructor.

→ Multiple Inheritance:

```
→  Class Base
   {
     public:
          Base()
          {
       cout<<" Base class constructor...\n";
          }
   };
     Class Base1
   {  public:
          Base1()
          {
       cout<<" Base1 Class constructor..\n";
          }
};                    ——————→ (Left to Right)
  Class Del: public Base1, public Base
  {
  }
```

```
public:
    Der()
    {
cout<<"Derived class Constructor...\n";
    }
};
void main()
{ clrscr();
  Der d;
}
```

O/P: Base1 Class Constructor.
Base class Constructor.
Derived class constructor.

In the above 2 programs the preference always will be takesplace for the base classes.

If the Baseclass contains a constructor, it could be called first followed by the Derived class.

When a Derived object is destroyed, it destructor called first, followed by Baseclass constructor, if it is exists in this Constructor functions are executed in the order of Derivation.

Destructor functions are executed in the reverse order of derivation.

Whenever constructor functions are executed in the order of derivation because a Baseclass has No Knowledge of any Derived class. Any initializations has to perform separatly.

So the <u>Constructors explicitly not participating in inheritence implicitly they are participating.</u>

<u>Passing Parameters to the Baseclass & to the Derived class:</u>

```
#include<iostream.h>
#include<conio.h>
class Base
{   protected:
        int a;
```

```
public:
        Base ()
      { a=0;
cout<<" Base clas Default constructor... \n";
      }
      Base(int x)
    {
        a=x;
    cout<<" Base clas palametrized constructor... \n";
      }
    void print ()
    {
     cout<<"A:"<<a<<endl;
    }
};
Clas der : public Base
{ protected:
            int b;
    public:
          Der() : Base (100)
        { b=0;          default
cout<<" Derived clas constructor... \n";
        }
        Der(int y)
      {
          b=y;
    cout<<"Derived clas palametrized constructor \n";
      }
    void display ()
{
 cout<<" B:"<< b<<endl;
};
```

```
Void main()
{
  clrscr();
  Dee d;
  d.print();
  d.display();
  getch();
}
```

O/P:

Baseclass parameterized constructor....
Derived class Default constructor....
A: 100
B: 0
Base class default constructor....
Derived class parametrized constructor
A: 0
B: 10

→ We have to pass explicitly, ⇒ Base class parametrized value either from derived class default (or) derived class Parametrized Constructor.

→ If we not pass any of the two it not raises any errors every time base class default constructor will invoke.

→ Whenever the constructors of the baseclass is existed when derived class of default & parameterized is present always first preference to Baseclass only.

If is important to understand the arguments to a baseclass constructor are passed via arguments to derived class Constructor.

Therefore always better to define baseclass arguments in Derived Class.

Ex: In the above prog

Dee(int y) : Base(100)

O/P: Base class parametrized
Derived class
Base class default-
Derived.

// Find the o/p of the progs.

→ 1. #Include<iostream.h>
#Include<conio.h>
Clan Base
{
Poblic:
~Base()
{
Cout<<"Base clan destructor!"<<endl;
}
};
Clan Derived: public Base
{
public:
~Derived()
{
}
Cout<<" Derived clan Destructor"<<endl;
}
};
Void main()
{
clscr();
Base *bptr = new Derived(); // Dynamically creating the object
delete bptr;
}

o/p:
Baseclass destructor.

→ 2. Clan MClan
{ public:
static int i=123;
};
Void main()
{
Coat<<" mclass::i"<<mClan::i;
}

3. →
```
class Test
{
public:
    void fun()
    {
cout<<"Inside Test::void fun()"<<endl;
    }
};
class Test_Der : public Test
{
public:
    void fun(int a)
    {
cout<<"Inside Test::void fun(int a)"<<endl;
    }
};
void main()
{
  ~~Test *ptr~~
  Test_Der test_Der;
  test_Der.fun();
}
```

---

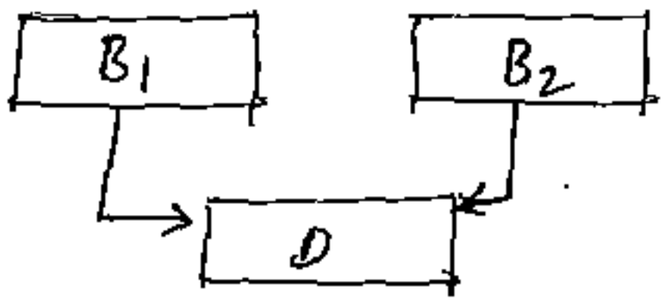
4. →
```
class Test
{
public:
    Test()
    {
cout<<"In Test()"<<endl;
    }
~Test()
{
cout<<"In ~Test()"<<endL
}
};
void main()
{
  Void * p = new Test; // Here Memory is allocated for void pointer
  delete p;            // p. So Destructor will not
}                      // invoke. only constructor invoked.
```

Here only Constructor is known.
If you want to invoke destructor
also Create the object with class name.

Test *p = new Test;

Multiple Inheritence

The Inheritence machinism in which a single derived clas will be Inherited from multiple Base clases, is called as multiple Inheritence



Granting Acces:

```
#Include <iostream.h>
#Include< conio.h>
Clan Base
{ public: Inf a:
};
clas Der:prirate Base
{ public:
        Base::a; // Granting Accen.
};
  Void main()
  {
    Clrscr();
    Der d;        O/P: 10
    d.a = 10;
cout<<'d.a;
    getch();
  }
```

→ When Base clan is inherited as private all public & protected members of that clan becomes privatemembers of the derived clan. However, in certine circumstances you may want to restore one (or) more Inherited members of their original accen Specification.

For example: You might want to grant certain public members of the base clas public Status in the derived clas even though

To

the Base class is inherited as private.

An Access declaration is takes this general form.

Baseclass :: member;

the Access declaration is put under the appropriate access heading in the derived class. You can use access declaration to restore the rights of public however the members declared as private & protected in base class can't be public by the derived class.

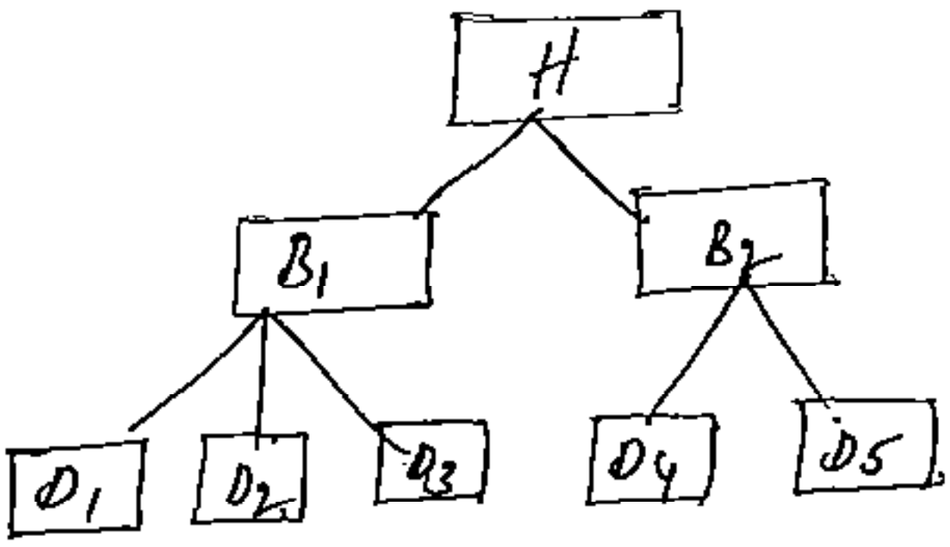
Granting Access for More than one class

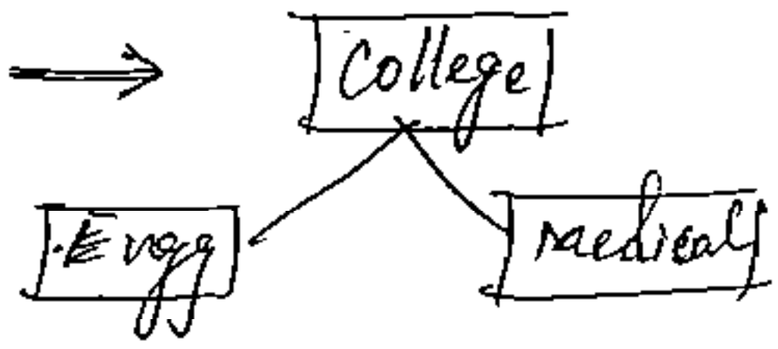
Example 2:

```
#include<iostream.h>
#include<conio.h>
class Base
{
  public:
        int a,b;
        Base()
        {
          a=10;
          b=20;
        }
};
class Derived1 : private Base
{
    public:
        Base::a;
        Base::b;
        void show()
        {
        cout<<"A:"<<a<<endl;
        cout<<" B:"<<b<<endl;
        }
};
class Derived2: public Derived1   // if it is Derived also private
{                                 //  Here D.show() becomes an errorr
    int sum;                      //  so // D.show();
    public:
        void calc()
        {
          sum = a+b;
        cout<<" SUM = "<<sum<<endl;
        }
};
void main()
{ clrscr();
  Derived2 D;
  D.show();
  D.calc();
  getch();
}
```

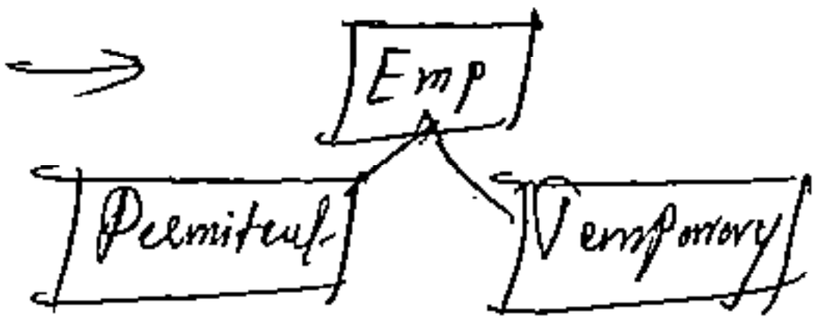
~~Hierarchi~~ Hierarchical Inheritence

The Inheritence Machanism in which multiple derived clases will existed from a single Base Class.





→ In this case we can use Switch case Mandatory for selecting any one College So. based on problem type of Changing the ~~Inheritence~~ way of writing the program.



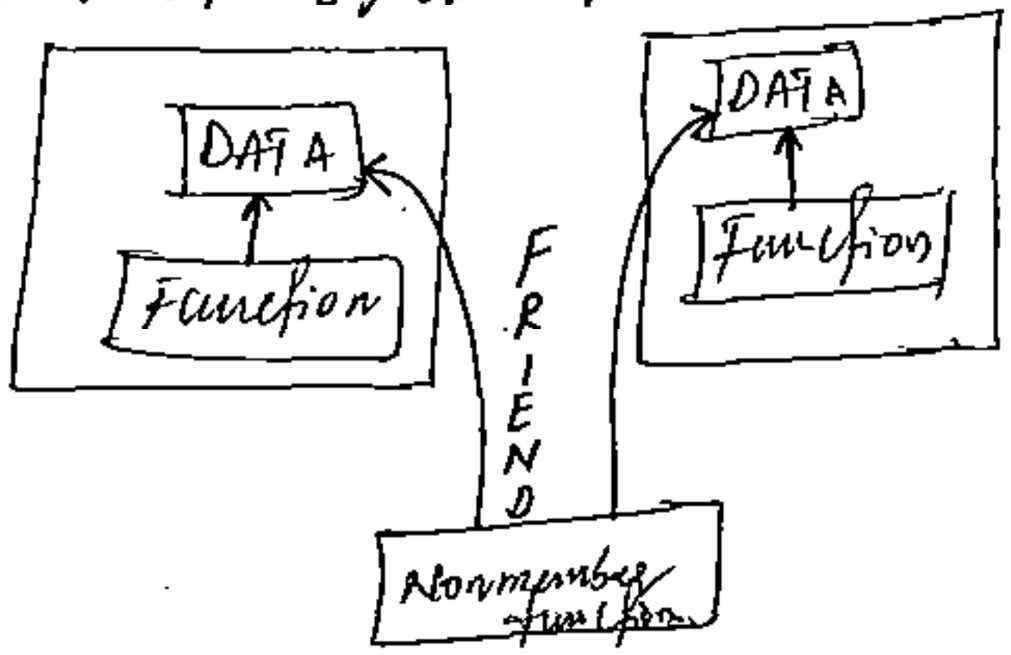

```
#include <iostream.h>
           <conio.h>
class emp
{
protected:
     int no;
     char name[15];
public:
     void accept();
     void show();
};
void emp :: accept()
{
  - cout<<"\n enter emp no:";
    cin>> no;
```

```
    cout<<"\n Enter Employee Name:";
    cin>> name;
    return;
}
    Void emp :: show()
    {
      cout<<"\n emp no :"<< no;
      cout<<"\n employee name:"<<name;
      return;
}
    Class parmenent: public emp
    { protected:
                float sal;
      public:   void accept();
                void show();
    };
    Void parmenent:: accept()
    { emp:: accept();
      cout<<"\n Enter salary:";
      cin>> sal;
      return;
    }
    Void parmenent:: show()
    { emp::show();
      cout<<"\n Salary :"<< sal;
      return;
    }
    Class Temporary: public emp
    { protected: float wage;
      public : Void accept();
               Void show();
    };
    Void Temporary :: accept()
    {
      emp:: accept();
      cout<<"\n Enter wage:";
      cin>> wage;
      return;
    } Void temporary :: show()
    { emp::show();
      cout<<"\n wages per day :"<<wage;
      return;
    }
      Void main()
    {
      int choice;
      do
      { clrscr();
        cout<<" Employee Maintenence System
      Cout<<"\n Main menu\n ------";
      Cout<<"\n 1. parmenent \n
              2. Temporary \n 0. exit";
      Cout<<"\n Enter ur Choice:";
      Cin>> Choice;                    F2
```

```
switch (choice)
{
case 1: permenent p;
        cout<<" Enter permenent employee details:";
                p.accept();
        cout<<" In display details";
        p.show();
        break;
case 2: temporary t;
        cout<<" Enter Temporary employee details:";
                t.accept();
        cout<<" Display details:";
                t.show();
                break;
        }
        getch();
}
while(choice>0 && choice<3);
}
```

FRIEND functions & Classes:

Sometimes a Private data of one class As to accen by the Non member, functions inorder to make a communication from b/w the Two clanes. That time of Situations the Friend function is useful to accen the private data & Public data.



Friend functions can be accen from outside the clan also.

→ In real world programing there could be a Situation were we would like two clanes to share a particular function.

* In C++, the common function can be made friendly with both clanes, which allows the function to have accen to the Private data of these clanes.

" These type of functions need not be a members of any of these clanes".

* To make an out-side function friendly to a clan. The Syntax is follows.

```
Class < Class Name>
{
  member;
  public: member;
      friend void < function name> (arguments);
};
```

" The function declaration should be preceded by the keyword friend.

" The function definition does not use either the keyword friend or scope operator ::.

" A function can be declared as a friend in any no. of clases.

" A friend function, though not a member function has full access rights to the members of the clas.

```
/* A Non member function Access the Private Data of the class */
#include<iostream.h>
#include<conio.h>
Class B; // forward declaration
Class A
{
    private:
        int a;
    public: void read()
        {
          cout<<"Enter the value:";
          cin>>a;
        }
        void write()
        {
          cout<<"A:"<<a<<endl;
        }
        friend void sum(A,B);
};
Class B"
{
  Private: int b;
  public:  void set()
        {
          cout<<"Enter the value:";
          cin>>b;
        }
```

```
    void get-b()
    {
    cout<<"B:"<<b<<endl;
    }
    friend void sum(A,B);
};
void sum(A obj1, B obj2)
{
    int S;
    S = obj1.a + obj2.b;
    cout<<"Sum:"<<S;
}
void main()
{
    Class cll;
    A obj1;
    obj1.read();
    obj1.write();
    getch();
    B obj2;
    obj2.set();
    obj2.get-b();
    getch();
    Sum(obj1, obj2); // Caller
    get-ch();
}
```

Friend function characteristics:

→ It is not in the scope of the class to which it has been declared as friend.

→ Since it is not in the scope of the class, it can't be called using object of that class.

→ It can be invoked like a normal function without the help of any object.

→ It can't access the member names directly and has to use an object name & dot membership operator with each membername.

→ It can be declared either in the public (or) private part of the class.

→ It has objects as arguments.

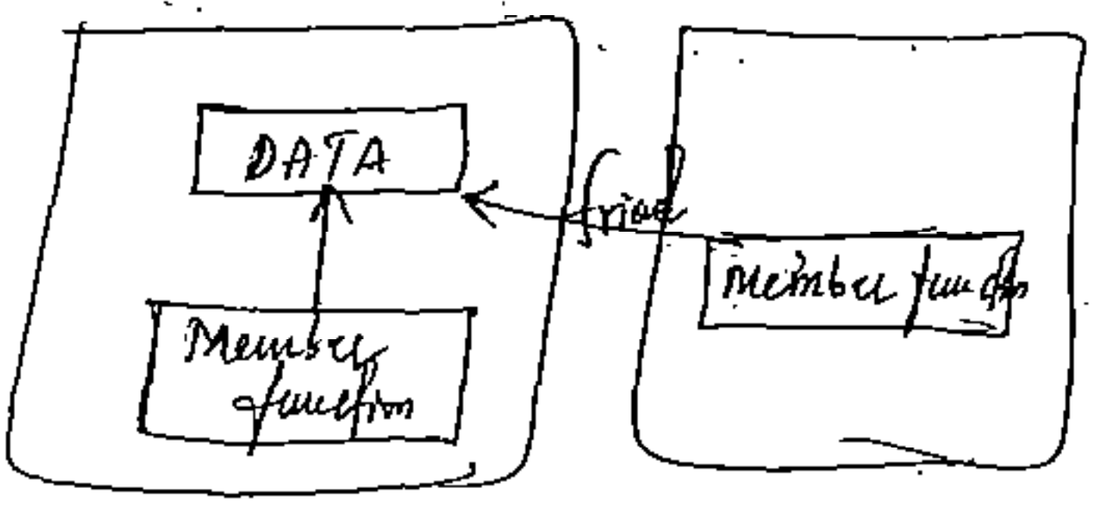
A Friend function can be called by reference.

→ In this, a pointer to the address of the object is passed & the called function directly works on the actual object used in the class.

→ The above method can be used to alter the values of the private members of a class.

```
Class husband
{  — —
   — —
};
Class wife
{  —
   —
};
```

/* a Member function of one class accessing the private data of another class. */



```
#include <iostream.h>
#include <conio.h>
#include <string.h>
 class stdper; // forward declaration.
 class stdmarks        pcr = personal.
{
    int m[3];
    char res[5];
public:  void read();
         void write(stdper);
};
  void stdmarks::read
  {
  cout<<" Enter 3 Marks \n";
   for(int i=0; i<3; i++)
   cin>> m[i];
   if( m[0] >= 35 && m[1]>=35 && m[2] >=35)
     strcpy (res, "pass");
   else
     strcpy ("res, "Fail");
}
 class stdper
 { int suo;
   char name[20];
public:  void read()
         {
          cout<<"suo:";
          cin>>suo;
          cout<<" sname:";
          cin>> name;
         }
 friend void stdmarks::write(stdper); // permission
};
```

```
void stdmarks :: write (stdper sree)
{
 cout<<" std no:" << sree.sno <<endl;
 cout <<"name :"<< name - sree . name <<endl;
 cout<<" marks \n";
   for(int i=0; i<3; i++)
     cout << m[i] <<endl;
 } cout <<" RESOLT:" << res <<endl;

 Void main()
{
   stdper s1;
   stdmarks s2;
 cout<<" calling read functions \n";
   s1. read();
   s2. read();
 cout <<" calling write function \n";
   s2 . write(s1);
   getches;
}
```

⟹ The Friend functions can also be apply in Inheritence Concept also.

Example of Friendfunction in th Inheritence

```
Clas Teacher;
Clas stud_teach; // forward declaration
Class student
{ protected:
        int no;
        Char name[15];
```

```
    int clan_study;
    float fees;
public:
    student() { }
    student(int x) { }
    void accept();
    void show();
    friend void fees_pf(student &, teacher &, stud_teach &, int &);
};
    void student :: accept()
{
    cout<<"\n Enter the student no:";
    cin>> no;
    cout<<"\n Enter the student name:";
    cin>> name;
    cout<<"\n Enter the class:";
    cin>> class_study;
    return;
}
    void student :: show()
    {
      cout<<"\n student no:";
  << cin>> no;
      cout<<"\n student name:";
    << cin>> name;
      cout<<"\n class";
  << cin>> class_study;
    cout<<"\n exam fees paid:"<< fees;
      return;
}
```

```
class teacher
{
protected: int no;
           char name[15];
           which_subject[15];
           float sal, pf;
public:
           void accept();
           void show();
  friend void fees_pf(Student &, teacher &, stud_teach &, int &);
};
void teacher::accept()
{
   cout<<"\n Enter the employee no:";
   cin>> ~~no~~ no;
   cout<<"\n Enter the Name of the teacher:";
   cin>> name;
   cout<<"\n enter the subject:";
   cin>> which_subject;
  cout<<"\n enter the salary:";
   cin>> sal;
   return;
}
  void teacher::show()
{
   cout<<"\n employee no:"<<no;
   cout<<"\n teacher employee name :"<<name;
   cout<<"\n Subject:"<<which_subject;
   cout<<"\n The Product toward:"<<pf;
}
```

```
class stud_teach : public student, public teacher
{ public: void accept();
          void show();
          friend void fees_pf(student &, teacher &, stud_tech &,
                                                    int &);
};
 Void stud_tech :: accept()
 {
  cout<<" \n Enter the student no:";
  cin >> student :: no;
  cout <<" Enter employee no:";
 cin>> teacher :: no;
 cout <<" \n enter his/her name :";
 cin >> student :: name;
  strcpy( teacher :: name, student :: name);
  cout<<" enter the class of study :";
 cin >> "\n student :: class_study;
 cout <<" \n enter subject teaching.";
 cin >>  which_~~sto~~ subject;
  cout <<" \n Enter salary:";
 cin >> sal;
 return;
 }
   Void stud_teach :: show()
{
 cout <<" \n student no :" << student :: no;
 cout <<" \n Employee no : << teacher :: no;
 cout <<" \n name :" << student :: name;
 cout <<" \n Class of study :" << student :: class_study;
 cout <<" \n Subject teaching :" << teacher :: which_subject;
 cout <<" \n Salary :" << sal;
```

```
cout<<"\n provident found is adjusted against the exam fees,
                Which is:"<<pf;
    Return;
}
    Void fees_pf(Student &s, teacher &t, Stud_teach &st, int &ch)
                                                          ↓
                                                        Choice
    {
      switch(ch)
       {
case 1: cout<<"\n enter exam fees:";
        cin>>s.fees;
        break;
case 2: cout<<"\n enter provident found:";
          t.pf = t.sal * 0.1;
        cout<<"t.pf;
        break;
case 3: Std st.pf = st.sal *0.1;
          st.fees = st.pf;
        break;
       }
    }

    Void main()
{
  int choice;
  do
  {
   clrscr();
   cout<<"\n School Maintenence System:";
   cout<<"\n 1. Student \n 2. teacher \n 3. student-teacher \n 0. exit";
   cout<<"\n Enter the choice [0-3]:";
   ciass choice;
   Student s;
   teacher t;
     stud_teach st;
```

```
switch(choice)
{
case 1: cout<<"\n Student Data entry form.";
        s.accept();
        fees_pf(s,t,st,choice);
        cout<<"\n displaying the details:";
        s.show();
        break;
case 2: cout<<"\n teachers data entry form.";
        t.accept();
        fees_pf(s,t,st,choice);
        cout<<"\n display the details.";
        t.show();
        break;
case 3: cout<<"\n student_teacher dataentry form.";
        st.accept();
        fees_pf(s,t,st,choice);
        cout<<"\n displaying the details:";
        st.show();
        break;
}
  getch();
} while(choice>0 && choice<=4);
}
```

Friend classes:

The Friend function can accept only attributes, but the Friend class can accept every thing in the class.

```
// Friend class
#include <iostream.h>
#include <conio.h>
Class A
{
public:  void fun1()
         {
         cout<<"public function...1n";
         }
protected:
         void fun2()
         {
         cout<<"protected function...1n";
         }
Private:
         void fun3()
         {
         cout<<"Private function....1n";
         }
         friend class B;
};
Class B
{
public:  void fun1()
         {
          A aobj;
         aobj.fun1();
         aobj.fun2();
         aobj.fun3();
         }
};
Void main()
{
   Class2();
```

→ Simply Friend is communication agent b/w the Two Classes.

Here A properties used by Class B, but 'B' properties can't used by 'A'. (because not declared in Class B as friend).

→ A Class if is Not possible to access the properties of B.

B obj:
obj-fun():
getch():

O/P: public function
protected function
Private function.

Note: For example friend function is removed in Class A. So it will be used in class B
'public' data will be accessed. but
protected & private raises an error.
because protected & private are not possible to accessed because not a friend.

→ Main() is also a friend function.
↳ It is not a class. it is a function.

Friend class with inheritence:

```
#include <iostream.h>
#include <conio.h>
Class Base
{
    friend class Der;
private:
        int value;
};
class Der : private Base
{
public:
        Der()
        {
            value=6;
        }
        void print()
        {
        cout<<"value:"<<value<<endl;
        }
};
void main()
{
    clrscr();
    Der d;
    d.print();
} getch();
```

→ This program raises an error without friend function. because private data can't inherit.
with the help of friend everything can be accessed.

O/P: value 6

→
```
#include<iostream.h>
#include<conio.h>
Class Base
{ Friend Class Der;
  Private:  int value;
};
Class Der : private Base
{
  public:
        Der()
        { value=6;
        }
};
Class Der1 : public Der // It raises an Error-
{ public:                          because 'Base' is only friend of
        Void Print()                'Der' class not Der1.
        {
        cout<<" value :"<<endl;
        }
};
Void main()
{ clrscr();
  Der1 d;
  d.Print();
  getch();
}
```

Notes: It raises an error. Since

'Base' is friend to 'Der'. but not for 'Der1'.

→ Friend is NOT inheritable

Applications

As a friend function planed by designe (i.e developa).

→ Device integrating is possible only by friend.

Ex: GPRS, Satelight-communications.

→ If is also used in application programming Interface (API)
Possible by friend class.
Ex: Money transfer one bank to another bank.

→ Unix system & Java never supports friend.

Accessing the private data without using Friend keywords By pointers

~~By pointers.~~

Hiding Vs Security

```
#include <iostream.h>
            <conio.h>
class A
{
    int x, y;
    public: int z;
};
void main()
{ clrscr();
  A obj;
  int *p;
  p=(int*)&obj;
  *p=10;
  p=p+1;
  *p=20;
  cout<<"*p;
  p=p-1;
  cout<<*p;
}
```

→ The protection of private data can be circumvented through pointers but this of course is cheating

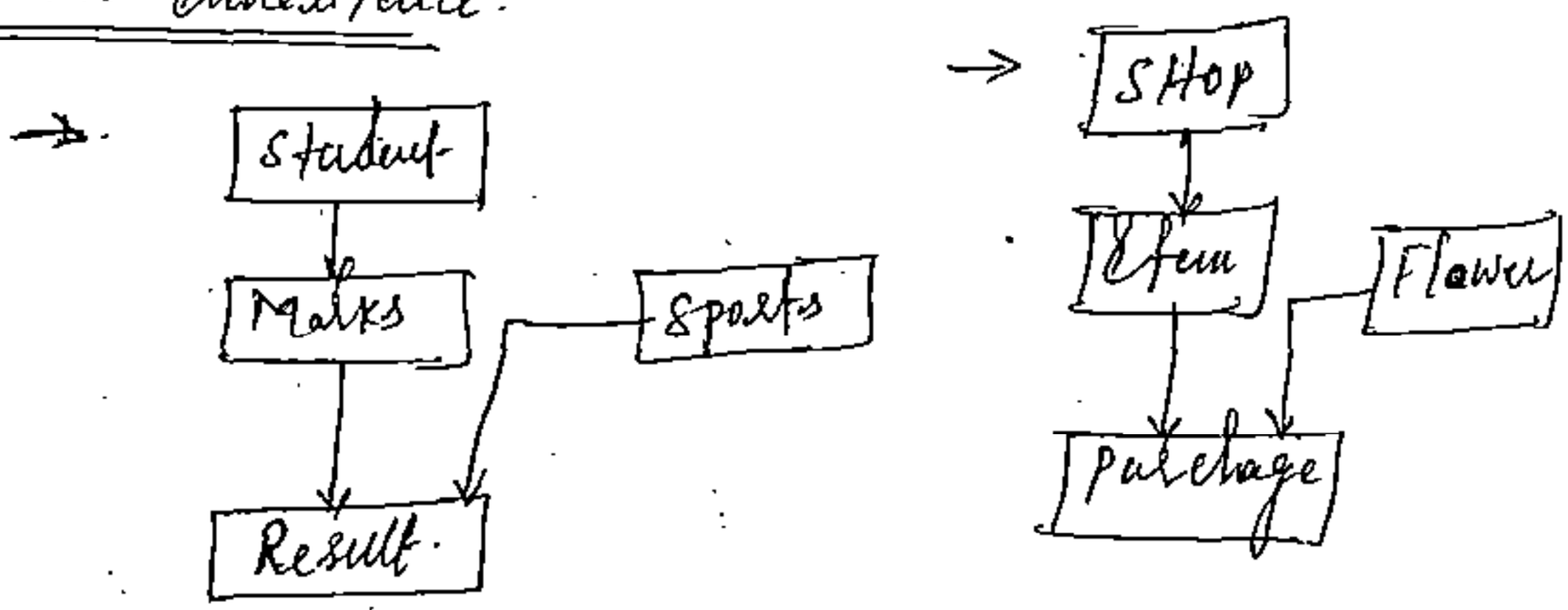
→ C++ protects against accident rather than deliberate fraud.

Bjarne Stroustrup.

Advantages of Inheritence:

1. Generic-to-specific way of thinking. (common)
2. Code reusability. (Object code reusability)
3. Fully tested & debugged code is used.
4. Increases program reliability.
5. Saves time and money.
6. Distribution of class Libraries is easy.
7. Source code is not required.
8. If don't touch the prewritten code

Hybrid Inheritence:



The Inheritence mechanism in which multiple representation models can be implemented with various interfacing is known as Hybrid Inheritence.

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

Class Shop
{ protected:
    int qty;
  Char shopname[15], name[15];
```

```
float price;
public:
        void accept()
        {
    cout<<"Enter the shop name:";
    cin>> shop name;
     return;
        }
      void accept-one()
     {
    cout<<"\n Enter the quantity purchaged:";
     cin>> qty;
    cout<<"\n Enter total price:";
    cin>> price;
     return;
     }
      Void show()
     {
     cout<<"\n quantity purchaged"<<qty;
     Cout<<"\n Total price"<< price;
      return;
     }
};
    Clan item: public shop
  {
     protected:
                  char size[15];
public:
          Void accept()
        {
    Cout<<"\n Enter the name of the item:";
    Cin>> name;
  Cout<<"\n Enter the size of the item:";
    Cin>> size;
     return;
        }
```

Nothing in this page -

```
        void show()
        {
    cout<<"\n Size of the item"<<c.size:
        return;
    }
};
    class flower
{
protected:
        char color[15], breed[15];
public:
        void accept()
        {
    cout<<"\n Enter the name of the flower breed:";
    Cin>> breed;
    Cout<<"\n Enter the color of flower:";
    Cin>> color;
    return;
    }
    void show()
    {
    Cout<<"\n Name of the flower"breed:"<< breed;
    Cout<<"\n Colour of the flower:"<< color;
    return;
    }
};
    Class purchage : public item, public flower
{
Protected: int item_type;
Public:
        Void accept();
        Void show();
};
```

```
void purchage::accept()
{
cout<<"In purchage entry form:";
cout<<"1. Stationary In 2. Flowers In 0. exit";
cout<<"In Enter Type of item purchage:";
cin>> item_type;
  Shop::accept();
  Switch(item_type)
  {
case1: item::accept(); break;
Case2: strcpy(name,"flower");
cout<<"In enter name of item: "<<item>name;
 flower::accept();
      break;
  }
Shop::accept_one();
  return;
}
 void purchage::show()
{
 cout<<"In The' "<<name<<" is/are purchaged from"<<shopname<<
          "its" " its details are:";
   Switch(Item_type)
   {
 Case1: Item::show(); break;
Case2: flower::show(); break;
}
   Shop::show();
     return;
}
```

```
void main()
{
  clrscr();
  ~~public~~
  publage p;
  p.accept();
  p.show();
}
```

Containce class (or) Compositional class:

```
class A
{ private:
  public: void xyz()
          {
          }
};
class B
{ private:
  public: void abc()
          {
          }
};
class c
{ private:
    Aaobj;
    B bobj;
  public: void muo()
          {
            aobj.xyz();
            bobj.abc();
          }
}
```

→ Machanism to make use of existing classes to define more & complex classes

→ The internal data (state) of one class contains another class. an object

→ By this we are providing Additional layer of data protection

```
void main()
{
  C cobj;
  cobj.muo();
}
```

→ The objects of another class acting as a private data member in another class.

→ Composition classes comes with grade deal of flexibility the member objects of your new class are usually private.

→ It makes inaccessable to the client-programmers.

Inheritence dont provide this type of flexibility. because of inheritence is so important in OOP. it is highly focused.

→ As the New programmer knows about inheritence concept- so we use inheritence.

Implentation through inheritence is orce complecated for this reason in real time we work with Composition Classes.

Polymorphism

Polymorphism

- **Compile time** Polymorphism (Static binding / Early binding)
  - Ex: Operator overloading, Function overloading
- **Run time** Polymorphism. (Dynamic binding / Late binding)
  - Ex: Virtual function.

Operator overloading:

The Need of Operator overloading.

→ Program To accept the distance traveled by 4 objects in Kilometers & meters & calculate total distance traveled by them.

```
#include <iostream.h>
        <conio.h>
 Struct dist
 {
    int Km, mts;
 };
  Void main()
 {
  clrscr();
  Dist d1, d2, d3, d4, d5;
  Cout<<"\n Enter the Distance of obj1";
  Cin>> d1.km >> d1.mts;
  Cout<<"\n Enter the Distance of obj2";
  Cin>> d2.km >> d2.mts;
```

```
cout<<"\nEnter the distance of obj3:";
cin>>d3.km>>d3.mts;
cout<<"\nEnter the distance of obj4:";
cin>>d4.km>>d4.mts;

  d5.mts=d4.mts+d3.mts+d2.mts+d1.mts;
  d5.km=d4.km+d3.km+d2.km+d1.km;
  while(d5.mts>=1000)
  {
    d5.km++;
    d5.mts=d5.mts-1000;
  }
 cout<<d5.km<<"\t"<<d5.mts;
 getch();
}
```

objects in
or Used by them

5/8/12

In the above pgm the manipulation of data members are manually integrated by i by individually refering to ~~the~~ through the objects. on Calculating the Sum of members or different types of objects as

d5.m1 = d4.m1 + d3.m1 + d2.m1 + d1.m1;

d5.km = d4.km + d3.km + d2.km + d1.km;

→ In this case the objects include only 2 members if the object includes multiple data members where the count may be in thousands that leads to inconsistency with abnormal representation of objects while Calculating the Sum.

→ In order to overcome the Specified drawback C++ Compiler Can be Supported with operator overloading (Internal & Externally)

Definition of operator Overloading :-

An operator which performs multiple Specified ~~values~~ to us along with a Creative nature is known as operator overloading

Operator overloading is an technique: enhance the power of extensability of C++.

C++ makes the user defined data types behave in much the same way ~~as~~ of built in times.

C++ has the ability to provide the operators with a Special meaning for a datatype.

mechanism of giving a Special meaning to an operator Can be Called as Operator Overloading..

when operator overloading is taken place only Existing operators which is Supported by C++ that operators only Can overload.

Operator overloading gives an additional meaning for arthimetic operator and many.

Operator overloading makes the programming logic more Convenient to handle.

Rules for overloading :-

→ Only Existing operators Can be overloaded, new operators Cannot been created.

→ The overload operator must have atleast one operand.

Of the a user defined time.

→ we Cannot change basic meaning of an operator.

→ overloaded operators will follow the rules of Syntax of the original operator.

→ All operators Cannot be overloaded

→ overloading Can be done for binary operators and unary operators

→ As we know that binary operator two operands
Unary operator only single Operand.

→ when we are overloading, we Can overload the operators by Converting by the member function and by implementing friend Keyword.

→ when we overloading Unary operators by member function it don't need any arguments (no arguments)

→ when we are overloading binary operators by member fns we Can overload by one argument needs

→ If it is friend fn unary operators. takes one argument

→ Binary operator needs 2nd argument if it is friend fn.

Syntax:

C
O
B
O
{
C
}

→ most p
will be
In thi

// Overloading

① #include
#include <
class umini
{
private :
Public :

logic more

new operator

one operand.

operator.

of Syntax

operators

operands

by single

erload

nber function

by member

no arguments)

by member

one argument

friend fn

Syntax :-

```
Return_type class_name :: Operator op(argument list)
{
    function body;
}
```

→ most probably the return type of a operator overloading will be the class type.

In this "op" indicates what operator we are overloading.

// Overloading of unary '-' operator.

```
① #include <iostream.h>
  #include <conio.h>
  class uminus
  {
  private :  int a;
  Public :
        Void read()
        {Cout<<" Enter the Value of a : ";
         Cin>>a;
        }
        Void write()
        {
        Cout<<"A: "<<a<<endl;
        }
        Uminus operator -()
        {
        (*this).a = -a;
         return *this;
        /* Uminus t;
           t.a = -a;
           return t;  */
  }
```

```
Void main()
{
clrscr();
Uminus U1,U2;
U1.read();
U1.write();
getch();
U2=-U1;
// translated as U2=U1.operator -();
U2.write();
getch();
}
```

```
class Uminus
{
public:
    void read();
    void write();
    void operator -();
};
void Uminus :: read()
{
cout<<"enter a,b,c:";
cin>>a>>b>>c;
}
void Uminus :: write()
{
cout<<a<<" "<<b<<" "<<c;
}
void Uminus :: operator -()
{
a=-a;
b=-b;
c=-c;
}
```

U2 = U1.operator -()
↑ object
↓ member function

return type
Uminus operator -()
↓ member function
{
}

Overloading ~~pre increment, Decrement~~ pre and post increment operators :-

---

```
class PreDec
{
private: int a;
public: void set()
        {
        cout<<"enter a:"; cin>>a;
        }
        void get()
        {
        cout<<"a:"<<a<<endl;
        }
```

6/8/14

obj's
//
obj

```
IncDec operator ++ ( )
{
 (*this).a = ++a;
  return *this;
}

IncDec operator --( )
{
  (*this).a = --a;
   return *this;
}
};

  Void main ( )
  {
   clrscr ( );
   Uminus U1,U2,U3,U4,U5;
   U1.read ( );
   U1.write ( );
   getch ( );
  U2 = ++U1;
    U3 = U1++;
    U4 = --U1;
    U5 = U1--;
    U1.write ( );
    U2.write ( );
    U3.write ( );
    U4.write ( );
    U5.write ( );
    getch ( );
  }
```

```
IncDec operator ++ (int k)
{
 (*this).a = a++;
  return *this;
}
```

// U1.operator++( )
// U1.operator ++(0)

6/2/24

```
obj3 = ob1 + ob2;
  // obj3 = obj1.operator + (obj2);
obj = obj1 + obj2 + obj3;
    obj1.operator + (obj2); + (obj3);
```

```
obj = obj1 + 10;
obj = obj1.operator + (
```

obj = 10 + obj1;
10.operator + (obj1)

Solution
Friend 87

· whenever operator overloading is happening the operator will be converted as member fn. and that member fn will be invoked by the object.

whenever the expression `obj3 = obj1 + obj2;` the left side of the operator object will invoke the member fn.

whenever the expression `obj = 10 + obj1;` the const no cannot possible to invoke the member fn. for this reason we overload the operator by using friend fn.

the expression obj = 10 + obj1; will works with friend fn. Since friend fn to invoke doesn't require an object.

Overloading insertion & Extraction operators :—

In the cranje of overloading insertion & Extraction of operators in between 2 objects we are using insertion & extraction operators already the insertion operator and extraction operators are overloaded right know we are providing for User defined object ie, erec

```
cin >> erec;
```

cin → object of istream
>> → friend function
erec → object of Emp

We provide the definition for the Extraction operator it invokes the definition and this objects will received by 2 classes with reference type and the formal parameter names can be changes as per the user.

whatever the data we are passing it is updating into the Erec;

C
C
C
C

7 | 8 | 12

Operator c

(1) class Nu
{
int n;
public:

```
/*
frend
/*or
fren
/*
fren
/*
fre.o
frie
frie
};
```

O.

operator will
will be invoked

left side of

we cannot
overload the

friend fn.

of operators
than operators
are
defined.

operator
allowed by
parameter

updating

Cout<<Cout; Cout<<&Cout; → Address of Cout
Cout<<Cin; → Address of Cin.

Cout.operator<<("hai");

7|8|12

Operator overloading using friend fn:-

```
(1) class Num
{
int n;
public: void getNum( )
        {
         Cout<<"Num:";
         Cin>>n;
        }
        void putNum( )
        {
        Cout<<"num: "<<n<<endl;
        }
    /* overloading arithmetic + operator */
  friend Num operator+(Num, Num);
  /* overloading relational > operator */
  friend int operator>(Num,Num);
  /* overloading Unary - operator */
  friend Num operator-(Num);
  /* overloading Compound operator += */
  friend void operator+=(Num&, Num);
  friend Num operator+(Num, int);
  friend Num operator+(int, Num);
};
```

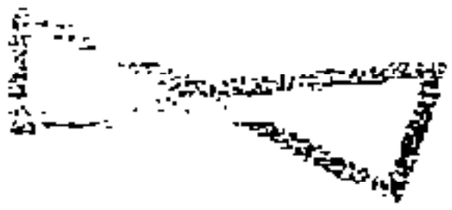
```
//Operator Overloading
http://kiransrinivas.wordpress.com
doubts mails to ykiransrinivas@yahoo.co.in

http://kiransrinivas.wordpress.com
doubts mails to ykiransrinivas@yahoo.co.in

#define W1 "SUN DAY"
#define W2 "MON DAY"
#define W3 "TUES DAY"
#define W4 "WEDNES DAY"
#define W5 "THURS DAY"
#define W6 "FRI DAY"
#define W7 "SATUR DAY"
#include<conio.h>
#include<iostream.h>
#include<string.h>

class DATE
{
  int dd;
  int mm;
  int yy;
  long dp;
  char week[30];
  void caldmy();
  void cal_dp_week();
  public:
  DATE();
  DATE(int,int,int);
  void getdate();
  void getdate(int,int,int);
  void showdate();
  DATE operator+(long);
  DATE operator-(long);
  long operator-(DATE);
  DATE operator-(long);
  DATE operator++();
  DATE operator++(int);
  DATE operator--();
  DATE operator--(int);
  int  operator>(DATE);
};
DATE::DATE()
{
  dp=1;
  caldmy();
}
DATE::DATE(int d,int m,int y)
{
  dd=d;
  mm=m;
  yy=y;
  cal_dp_week();
}

int DATE::operator>(DATE D)
{
  return dp>D.dp;
}
```

```
62
63  DATE DATE::operator++()
64  {
65   dp++;
66   caldmy();
67   return *this;
68  }
69  DATE DATE::operator++(int)
70  {
71   DATE td;
72   td=*this;
73   dp++;
74   caldmy();
75   return td;
76  }
77  DATE DATE::operator--()
78  {
79   dp--;
80   caldmy();
81   return *this;
82  }
83  DATE DATE::operator--(int)
84  {
85   DATE td;
86   td=*this;
87   dp--;
88   caldmy();
89   return td;
90  }
91
92  DATE DATE::operator-(long days)
93  {
94   dp=days;
95   caldmy();
96   return *this;
97  }
98  long DATE::operator-(DATE &D)
99  {
100  return dp-D.dp;
101 }
102 void DATE::getdate(int d,int m,
103                    int y)
104  {
105   dd=d;
106   mm=m;
107   yy=y;
108   cal_dp_week();
109  }
110 DATE DATE::operator+(long days)
111 {
112   DATE td;
113   td.dp=dp+days;
114   td.caldmy();
115   return td;
116 }
117 DATE DATE::operator-(long days)
118 {
119   return operator+(-days);
120 }
```

total days passed.

This fn will assign the values for days months & years and calculate based on the year how many total days passed.

```
void DATE::caldmy()
{
 long int tdp=dp;
 for(yy=1;tdp>365;tdp-=365)
  {
     if(tdp==366 && yy%4==0)
     break;
     if(yy%4==0)
     tdp--;
     yy++;
  }

  if(tdp>=1&&tdp<=31)//Jan
  {
     mm=1;
     dd=tdp;
  }

  if(tdp>=32&&tdp<=59) //Feb
  {
     mm=2;
     dd=tdp-31;
  }

if(yy%4==0 && tdp==60) //leap
  {
     mm=2;
     dd=tdp-3
  }
  if(yy%4==0
  tdp--;

  if(tdp>=60&&tdp<=90) //mar
  {
     mm=3;
     dd=tdp-59;
  }
  if(tdp>=91&&tdp<=120)
  {
     mm=4;
     dd=tdp-90;
  }
  if(tdp>=121&&tdp<=151)
  {
     mm=5;
     dd=tdp-120;
  }
  if(tdp>=152&&tdp<=181)

  {
     mm=6;
     dd=tdp-151;
  }
  if(tdp>=182&&tdp<=212)
  {
     mm=7;
     dd=tdp-181;
  }
  if(tdp>=213&&tdp<=243)
  {
     mm=8;
     dd=tdp-212;
```



```
    }
    if(tdp>=244&&tdp<=273)
    {
       mm=9;
       dd=tdp-243;
    }
    if(tdp>=274&&tdp<=304)
    {
       mm=10;
       dd=tdp-273;
    }
    if(tdp>=305&&tdp<=334)
    {
       mm=11;
       dd=tdp-304;
    }
    if(tdp>=335&&tdp<=365)
    {
       mm=12;
       dd=tdp-334;
    }
http://kiransrinivas.wordpress.com
doubts mails to vkiransrinivas@yahoo.co.in

       switch(dp%7)
       {
       case 1:strcpy(week,W1);
        break;
     case 2:strcpy(week,W2);
        break;
     case 3:strcpy(week,W3);
            break;
     case 4:strcpy(week,W4);
        break;
     case 5:strcpy(week,W5);
        break;
     case 6:strcpy(week,W6);
        break;
     case 0:strcpy(week,W7);
            break;

       }
}// END FOR CAL DMY

void DATE::cal_dp_week()
{
 int syy;
 dp=0;
    for(syy=1;syy<yy;syy++)
 {
     dp+=365;
     if(syy%4==0)
     dp++;
 }
switch(mm)
{
     case 12:dp+=30; //Nov
     case 11:dp+=31;
     case 10:dp+=30;
     case 9:dp+=31;
     case 8:dp+=31;
```

```
        case 7:dp+=30;
        case 6:dp+=31;
        case 5:dp+=30;
        case 4:dp+=31;  //mar
        case 3:dp+=28; //Feb
        case 2:dp+=31; //jan
        case 1:dp+=dd;
    }
    if( yy%4==0 && mm>2)//for leap
        dp++;
        switch(dp%7)
        {
    case 1:strcpy(week,W1);
         break;
    case 2:strcpy(week,W2);
                break;
    case 3:strcpy(week,W3);
                break;
    case 4:strcpy(week,W4);
                break;
    case 5:strcpy(week,W5);
                break;
    case 6:strcpy(week,W6);
                break;
    case 0:strcpy(week,W7);
         break;
         }
} // END FOR CAL DP &WEEK

 void DATE::getdate()
{
     cout<<"\nENTER DD:";
     cin>>dd;
     cout<<"\nENTER MM:";
     cin>>mm;
     cout<<"\nENTER YY:";
     cin>>yy;
     cal_dp_week();
}

 void DATE::showdate()
 {
  cout<<"\nDATE:"<<dd
    <<"/"<<mm<<"/"<<yy;
   cout<<"\nDP ="<<dp;
   cout<<"\nWEEK ="<<week;
 }

DATE operator+(long days
               ,DATE D)
{
    return D+days;
}
void main()
{
  DATE d1,d2,d3,d4;
  clrscr();
  d1.getdate(23,2,1981);
  d1.showdate();
  getch();
}
```

7,22,700 + 495 = 7,23,195 + 31 + 23 = 7,23,249

Date: 23/2/1981
DP: 7,23,249
WEEK: MONDAY

```
    d2.getdate(15,3,2005);
    d2.showdate();
    getch();

  cout<<"D2 is Greater than D1 return 1 or 0 :"<< (d2>d1);
   /* cout<<endl<<"D1 Data";
    d1.showdate();
    getch();*/
    d2=d1+365+1;
     cout<<endl<<"D2 Data";
     d2.showdate();
     getch();
   d2=d1-365;
     cout<<endl<<"D2 Data";
   d2.showdate();
   getch();
   cout<<endl<<d2-d1;
   getch();
   d3= d1-d2;
  cout<<endl<<"D3 Data";
   d3.showdate();
   getch();
   d3.getdate(28,2,2009);
   d4=d3++;
     cout<<endl<<"D3 Data";
  d3.showdate();
  getch();
   cout<<endl<<"D4 Data";
  d4.showdate();
  getch();
  }
```

7,31,460 + 501 + 31 + 28 + 15 = 7,32,035

Dec; - 15/3/2005

DP = 7,32,035

Weeks Tuesday.

d2. Operator > (d1);
1
memfn.

d1. operator + (365);

+ operator fn invoked 2 times.

d1. operator - (365);

d2. operator - (d1);

operator ++ (0);

DATE : 2/1981

DP : 7 9

WEEK = MON DAY

DATE : 15/3/2005

DP = 732035

WEEK = TUES DAY D2 is Greater than D[illegible]eturn 1 or 0 : 1

D2 Data

DATE : 24/2/1982

DP = 723615

WEEK = WEDNES DAY

D2 Data

DATE : 24/2/1980

DP =

```
        obj3=obj1+obj2;
obj3.display("Your full name
:");
        cout<<"\n";
        obj3.show();

        if(obj1>obj2)
        cout<<" >";
        else If(obj1<obj2)
        cout<<"<";
        else
        cout<<"==";
        obj2.show();
        getch();
        }
```



```
#Include<iostream.h>
#Include<conio.h>
class Time
 {
  int hrs;
  int mts;
  int secs;

 void calculate()
 {
  if(secs>=60)
   {
       mts+=secs/60;
       secs%=60;
   }
  if(mts>=60)
   {
       hrs+=mts/60;
       mts%=60;
   }
   if(hrs>=12)
   hrs=hrs%12;
 }

 public:
 void gettime();
 void showtime();
 void sum(Time&,Time&);
 Time sum(Time&);
 Time operator+(Time&);
 operator long int();
 Time operator++();
 Time operator++(int);

};
```

```
void Time:: gettime()
 {
  cout<<"ENTER HRS :";
  cin>>hrs;
     cout<<"ENTER MINTS :";
     cin>>mts;
     cout<<"ENTER SECS :";
     cin>>secs;

 }
void Time::showtime()
 {
   calculate();
   cout<<"\n Time=
"<<hrs<<":"<<mts<<":"<<s
ecs;

 }


 void
Time::sum(Time&A,Time&B)
 {
   hrs=A.hrs+B.hrs;
   mts=A.mts+B.mts;
   secs=A.secs+B.secs;
 }

Time Time::sum(Time & A)
 {
  Time Tt;
  Tt.hrs=hrs+A.hrs;
  Tt.mts=mts+A.mts;
  Tt.secs=secs+A.secs;
  //Tt.caluclate();
  return Tt;
 }
Time
Time::operator+(Time&A)
 {
  Time Tt;
  Tt.hrs=hrs+A.hrs;
  Tt.mts=mts+A.mts;
  Tt.secs=secs+A.secs;
  return Tt;
 }


 Time ::operator long int()
 {

return(hrs*60*60)+(mts*60)
+secs;
  }

 Time Time::operator++()
  {
  Time Tt;
  Tt.hrs=++hrs;
  Tt.mts=++mts;
  Tt.secs=++secs;
  return Tt;
  }

Time Time:: operator++(int)
```

```
{
 Time Tt;
 Tt.hrs=hrs++;
 Tt.mts=mts++;
 Tt.secs=secs++;
 return Tt;
}


void main()
 {
 clrscr();
  Time T1,T2,T3,T4,T5;
  long int ts;
 // T1=sum(T2,T3);
   cout<<"\nEnter t2
object\n";
   T2.gettime();
   T2.showtime();
   cout<<"\nEnter t3
object\n";
   T3.gettime();
   T3.showtime();
    getch();

   T1=sum(T2,T3);
   cout<<"\nt1 Data\n";
   T1.showtime();
   cout<<"\nt1 Data\n";
   T4=T2.sum(T3);
   T4.showtime();
   T5=T2+T3;
   T5.showtime();

   T4=T5++;
   T4.showtime();
   T5.showtime();

   T4=++T5;
   T4.showtime();
   T5.showtime();



    ts=T5;
   ts=T1.operator long int();
   cout<<"\nTime in Seconds
: "<<ts<<endl;
 }
```

```
//Operator Overloading
http://kiransrinivas.wordpress
.com
doubts mails to
vkiransrinivas@yahoo.co.in

/*OVERLOADING UNARY
INCREMENT OPERATOR*/
#include<iostream.h>
#include<conio.h>
class Num
{
 int n;
public:
 void read()
 {
  cout<<"n : ";
  cin>>n;
 }
 void write()
 {
  cout<<"n : "<<n<<endl;
 }
 /*Post Increment*/
 Num operator ++(int k)
 {
  Num t;
  t.n=n++;
  return t;
 }
 /*Pre Increment*/
 Num operator ++()
 {
  Num t;
  t.n=++n;
  return t;
 }
};
void main()
{
 Num n1,n2;
 cout<<"enter n1\n";
 n1.read();
 cout<<"n1\n";
 n1.write();
 getch();

 cout<<"n1++\n";
 n1++;
 n1.write();
 getch();

 cout<<"++n1\n";
 ++n1;
 n1.write();
 getch();

 cout<<"n2=n1++\n";
 n2=n1++;
 cout<<"n1\n";
 n1.write();
 getch();
 cout<<"n2\n";
 n2.write();
 getch();
 cout<<"n2=++n1\n";
 n2=++n1;
 cout<<"n1\n";
 n1.write();
 getch();
 cout<<"n2\n";
 n2.write();
 getch();
}

http://kiransrinivas.wordpress
.com
doubts mails to
vkiransrinivas@yahoo.co.in
//Overloading insertion and
extraction operators
#include<iostream.h>
#include<conio.h>

class Emp
{
 int empno;
 char name[20];
 float sal;
public:
 friend ostream & operator
 <<(ostream &, Emp &);
 friend istream & operator
 >>(istream &, Emp &);
};
ostream &operator
<<(ostream &out, Emp &e)
{
 out<<"Empno :
 "<<e.empno<<endl;
 out<<"Name :
 "<<e.name<<endl;
 out<<"Sal :
 "<<e.sal<<endl;
 return out;
}
istream & operator
>>(istream &in, Emp &e)
{
 cout<<"Empno : ";
 in>>e.empno;
 cout<<"Name : ";
 in>>e.name;
 cout<<"Sal : ";
 in>>e.sal;
 return in;
}

void main()
{
 Emp erec;
 cout<<"Enter Emp
details\n";
 cin>>erec;
 cout<<"The employee details
are\n";
 cout<<erec;
 getch();
}

//Overloading + Operator for
strings
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string
{
char *str;
public:
        string()
        {
        str=new char;
        str[0]='\0';
        }
        string (char *msg)
        {
        cout<<msg;
        cin>>str;
        }
        void display(char
*msg)
        {
        cout<<msg<<str;
        }
string operator +(string obj2)
        {
        string temp;
        temp.str=new
char(strlen(str)+strlen(obj2.st
r)+1);
        strcpy(temp.str,str);
        strcat(temp.str,obj2.s
tr);
        return(temp);
        }
        void show(void)
        {
        cout<<str;
        }
        int operator > (string
obj2)
        {
        if(strcmp(str,obj2.str)
>0)
        return 1;
        return 0;
        }
        int operator < (string
obj2)
        {
        if(strcmp(str,obj2.str)
<0)
        return 1;
        else
        return 0;
        }
        };

        void main()
        {
        clrscr();
string obj1("Enter your first
name:");
string obj2("Enter your
second name:");
        string obj3;
```

7/8/17

Operator Overloading using Friend function.

```
#include<iostream.h>
#include<conio.h>
Class Num
{
  int n;
public:
      Void getNum()
      {
        cout<<"Num:";
        cin>>n;
      }
      Void putNum()
      {
       cout<<"Num":<<n<<endl;
      }
/* overloading unary-operator */
   friend Num operator - (Num);
/* Overloading arthematic + operator */
   friend Num operator + (Num, Num);
/* Overloading relational operator > */
   friend int operator > (Num, Num);
/* Overloading Compound operator += */
   friend Void operator += (Num, Num);
   friend Num operator + (Num, int);
   friend Num operator + (int, Num);
};
Num operator - (Num n1)
{
   Num t;
   t.n = -n1.n;
    return t;
}
```

```
Num operator +( Num obj1, Num obj2)
{   Num temp;
    temp.n = obj1.n + obj2.n;
    return temp;
}
int operator>(Num n1, Num n2)
{   if(n1.n > n2.n)
      return 1;
    else
      return 0;
}
void operator+=( Num &n1, Num n2)
{   n1.n = n1.n + n2.n;
}
Num operator +(Num obj1, ints)
{   Num temp;
    temp.n = obj1.n + s;
    return temp;
}
Num operator +(ints, Num obj2)
{   Num temp;
    temp.n = s + obj2.n;
    return temp;
}
void main()
{   clrscr();
  Num n1, n2;
Cout << " Entre Nombres : \n";
```

```
n1.getNum();
n2.getNum();

Num n3;
n3 = -n1;
cout<<"n3 = -n1 :"<<n3;
n3.putNum();
getch();
n3 = n1+n2;
cout<<"n3 = n1+n2 :"<<n3;
n3.putNum();
getch();
n1+=n2;
cout<<"n1+=n2 :"<<n1;
n1.putNum();
getch();

if(n1>n2)
cout<<"n1 is big";
else
cout<<"n2 is big";
getch();

n3 = n1+10;
cout<<"n3 = n1+10 :"<<n3;
n3.putNum();
getch();
n3 = 10+n2;   10+n2
cout<<"n3 = 10+n2 :"<<n3;
n3.putNum();
getch();
}
```

O/P:

```
Num: 5
Num: 8
n3 = -n1
num = 5
n3 = n1+n2
num = 13
n1+=n2
num = 13
n1 is big
n3 = n1+10
num = 23
n3 = 10+n2
num: 18
```

→ Some operators can't possible to overload by using friend functions like functioncall (), [], → pointer to memberaccess.

→ Some operators not possible to overload with both

- Member selection.
- .* pointer to member Selection
- :: Scope resolution operator
- ?: Teliinary operator

→ The operator "." and "*" are means of refering Structure (or) Class members. They take name rather than a value as their second operand. For Ex When we say p.a is the value. Since to an overload operator function we must pass a value & not the name we are not allowed to overload the "." operator a = membervariable

→ The "::" operator refers to a variable rather than a value. Hence it can't be Overloaded.

→ Even though ?: are operators they do not perform an operating as such. They offer a compact way of representing a Simple if-else. Hence they can't be overloaded.

p.5 // Not possible ::5 // Not possible

p.a // possible ::a // possible.

5+2 // possible

Type Conversions

Type conversion will be takes place from user defined types.
There are diff Types of Conversion will take place

1. Basic to class type.
2. Class to Basic type.
3. Class to Class.

Basic to class:

This is one type of Conversion will be takes place by using Constructors.

When this Conversion will taking place we define one parameterised constructor

```
Class Num
{ int a;
public: Num (int c)
        {
          a = c;
        }
void print()
{
  cout << a << x;
  cout << a;
}
};
```

```
m
num ()
{ a = 0; }
```

```
void main()
{
  num n;
  n = 10;
  n.print();
}
```

Class to Basic:

In This type of Conversion to Convert we require Operator Overloading Concept explicitly by the User.

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

```
Clan Nom
{ Inf 6;
public:
      num()
      {
        6=50;
      }
 Operator Inf()
 {
      return 6;
 }
      Void Show()
      {
    Cout<<"6 is "<<6;
 }; }
```

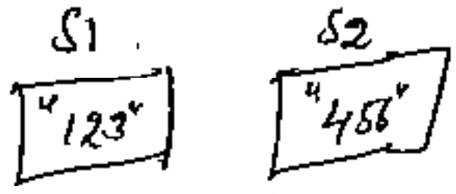
```
Void main()
{
   Clas s();
   num n;
  Inf c=n; //
  Cout<<c;
}
```

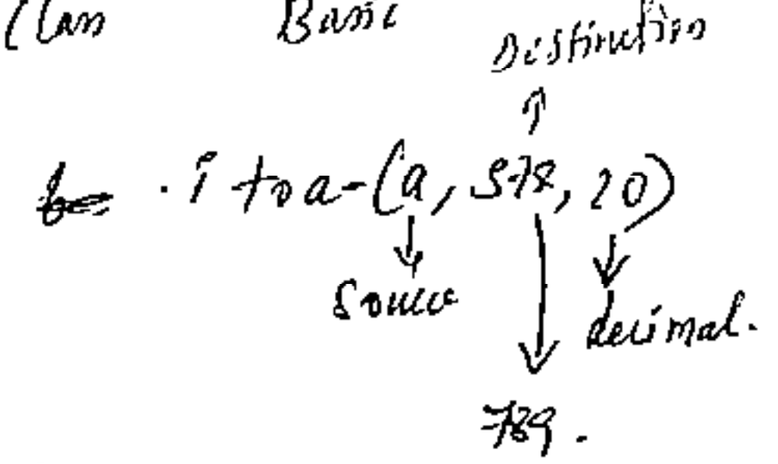
O/P: 50.

Example 3: basic to Clan, Clas to basic.

```
#Include <iostream.h>
#Include <conio.h>
#Include <string.h>
#Include <stdlib.h>
 Clan String
{ private: Char str[20];
 public: String()
        {
          str[0]='\0';
        }
 String (Char*s)
{
  Strcpy(str,s);
}
 String (Inf a) // for conversion
{ ito a (a, str, 10);
}
```

```
Operator int()
{
 int l, ss=0, k=1;
  l=strlen(str)-1;
  while (l>=0)
  {
    ss = ss+ (ss[l]-48) * k;
     l--;  *k=10;
  }
    return ss;
  void print()
  {
  cout<<"str<<endl;
  }
};
   void main()
   {
     clrscr();
   String s1("123");
   String s2("456");
   s1. print(); //"123"
   s2. print(); //"456"
   getche();
   String s3 = 789;
   s3. print(); //"789"

   s1=150;
   int l= int(s2);
   cout<<i<<end; //456
     i = s3;
   cout<<i; //789
}
```

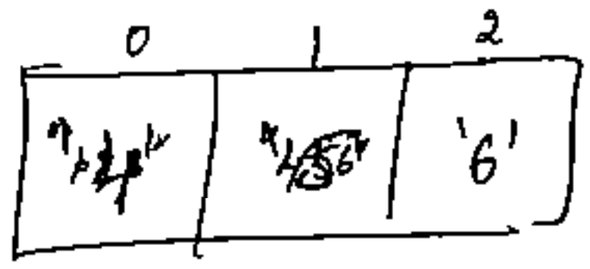




int i = int(s2)
↳ "456"

l=0, ss=0, K=1

l=3



l = 3-1

l = 2

s>=0 / 1>=0 / 0>=0

s3 = ss+(str[l]-48)*k
= 0 +(54-48)*k
= 0+6*1
= 6

= 6+(str[1]-48)*10;
= 6+(53-48)*10
= 56

= 56+(str[0]-48)*100
= 56+(52-48)*100
= 56+4*100
= 456

Class to Class

When this conversion is taking place either we can implement by the Operator Overloading (or) by the Parameterized constructor.

// User to User

Conversion Routine in Source class.

```
class date
{
private:
    char dt[9];
public:
    date()
    { dt[0]='\0';
    }
    date(char *s)
    {
      strcpy(dt,s);
    }
    void show()
    {
  cout<<"dt"<<endl;
    }
};

class dmy
{
private:
    int mm,dd,yy;
public:
  dmy(int d,int m,int y)
  { dd=d; mm=m; yy=y;
  }
  operator date()
  {
  char temp[3], str[9];
   itoa(dd,str,10);
   strcat(str,"/");
   itoa(mm,temp,10);
   strcat(str,temp);
   strcat(str,"/");
   itoa(yy,temp,10);
   strcat(str,temp);
   return (date(str)); // calling
                       previous class parameterized constructor
  }
};

void main()
{
  date d1;
  dmy dmy d2(17,11,94);
    d1=d2;
} d1.show();
}
```

→ Conversion Coufinge in Destination Clan

```
#include <iostream.hs
          <conio.hs
Clan math
{
  Public: inta,b;
   match()
   {
     a=10;
     b=10;
   }
};  }

Clan num
{
   int a,b;
  Poblic: num()
          {
            a=100;
            b=50;
          }

num(math m) // conversion coufine in destination clan.
  {
     a= m.a;
     b= m.b;
  }
  Void show()
{
cout<<"a is:"<<a<<"b is:"<<b<<endl;
}
};
  Void main()
{  Clasc x();
   num n;
   math m;
  n=m;
   n.show();
}
```

→ Clan dmy
{
private: int dd, mm, yy;
public:
dmy(int d, int m, int y)
{
dd = d; mm = m; yy = y;
}
int get-day()
{
return day;
}
int getmth()
{
return mth;
}
int getyr()
{ return yr;
}; }
Clan date
{ private: char df[8];
public: date()
{ df[0] = '\0';
}
date(char *s)
{ strcpy(df, s)
}
date(dmy, t) // Conversion routine in destination clas -
{ get-day();
int d = t-get-day();
int m = t-get-mth();
int y = t-get-yr();
// get-mth(), get-yr();
Store d, m, y in df[]

itoa(d, df, 10);
strcat(df, "/");
itoa(m, df, 10);
strcat(df, "/");
itoa(y, df, 10);

void show()
{
cout << df << endl;
}
};
void main()
{
date d1;
dmy d2(17, 11, 98);
d1 = d2;
d1.show();
}

V-Imp

~~Conversion~~ 12 Class - Class.

1. Create a Class of Conversion time 24 hours to 12 hours. (OR)

2 4 12 4 24 4

Runtime polymorphism

What is the need of Runtime polymorphism.

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

class father
{
  ~~public~~:
    char name[20];
    public: father(char *fname)
            {
              strcpy(name, fname);
            }
            void show()
           {
    cout<<"fathername:"<<name<<endl;
           }
};
  class son: public father
{   char name[20];
    public: son(char*sname, char *fname): father(fname)
            {
              strcpy(name, sname);
            }
            void show()
           {
    cout<<"son name:"<< name <<endl;
           }
};
```

```
void main()
{
  clrscr();
  father *fp;
  father f1("scott");
  fp = &f1;
  fp -> show();
  son s1("clark", "scott");
  fp = &s1;
  fp -> show();
}
```

→ In the above program at the Two times of passing baseclass object & Derivedclass objects Only the Baseclass member function of show is Invoking it doesn't Invoking Derived class memberfunction.

Since Compiletime polymorphism is taking place To overcome this problem & to invoke derived member-function we have Virtual functions.

⟹ The Realprogramming of Objectoriented takesplace Using polymorphism through virtual function.

Overloading of functions, (or) operators are also considered as polymorphism but it is Compile time polymorphism.

In order to implement Runtime polymorphism We need of virtual functions.

When virtual functions are used a program that apears tobe calling a function of one class ~~main~~ may in reality be calling a function of diff class

Virtual function Requirments:

Functions from diff classare executed through the Same function call.

The Derived clases almost be derived from the Same baseclases.

The Base must contain functions which have been declared virtual.

The pointers to objects of a derived clas are type compatable with pointers.

Taking the addres of the derived clas objet & treating it as a the addres of the base clas objet, it is called as Up casting.

By defining virtual functions to youe function the Same function call executes diff functions depending on the contents of pointer.

Whenevere the functions in a normalway during compilation it is called early binding.

At the Same time if the pointer knows about the objet it is pointing the appropriate function, get's called which is refined as Late binding (or) dynamic binding

Dynamic binding throough objet Typecasting

We Can also acheive dynamic binding by Objet type casting without using the virtual functions.

This typecasting has to be takes place for the Base clas objet.

need also derived class object.

If the Type casting is not takenplace for the Base class object still it can invoke exact member function. but derived class object must be type casting.

```
#include <iostream.h>
        <conio.h>
Class A
{
protected:
  int i;
public: void accept()
      {
      cout<<"\n Enter any number:";
      cin>> i;
      }
  void show()
  {
  cout<<"\n number"<<i;
  }
};
  Class B: public A
  {
  protected:
    float f;
  public: void accept()
        {
        cout<<"Enter any float:";
        cin>>f;
        }
    void show()
    {
    cout<<"\n float number"<<f;
    }
};

Class C : public B
{ protected:
    char name[15];
public :
    void accept()
  {
  cout<<"\n Enter UR good Name:";
  cin>> name;
  }
  void show()
  {
  cout<<" Hai"<<name<<"!";
  }
};

  void main()
{
    A *obj;
  int choice;
do
{ cout<<"\n Call object of";
cout<<"\n 1. Class A \n 2. Class B \n
    3. class C \n 0. Exit";
cout<<"\n Enter UR choice!";
 cin>> choice;
 switch(choice)
{
 case 1: obj -> accept();
      obj -> show();
      break;
case 2: ((B*)obj) -> accept();
      ((B*)obj) -> show();
      break;
```

```
Case 3:  ((*C) obj) -> accept();
         ((*C) obj) -> Show();
              break;
    } getche();
  } while (choice > 0 && choice < 4);
}
```

→ Object Type-casting can also be implemented for the dynamic objects.

Using Virtual Keyword:

A Virtual function is a member function, that is declared in the Base class and is redefined by derived class.

To create a virtual function, precede the function declaration in the baseclass with the keyword virtual.

When a class containing a virtual function is inherited, the derived class.

Virtual function implement the "Single method: Multiple implementations" paradiam intrinsic to polymorphism.

The Virtual function with the base class defines the form of the interface to the function.

Each redefined of the Virtual function by a derived-class implements it's operation as it relates specifically to the derived class. - That is redefinition creates a specific method.

When a base clas pointer points to the Derived clas objects that contains a virtual function. C++ determines which version of that function to call based upon the type of the object pointed by the pointer.

And this determination made at runtime. Thus, when diff ~~types of~~ derived clas objects are pointed to by the base clas pointer at different points of time, diff versions of the virtual function are executed.

Example by using virtual Keyword:

56

```
#include<iostream.h>
        <conio.h>
Class Base
{
public:
        virtual void vfunc()
        {
cout<<"\n Base clas vfuncs\n";
        }
};
Clas Derived: public Base
{ public:
        Void vfunc()
        {
cout<<" Derived clas vfuncs\n";
        }
};
Clas Der1: public Derived
{ public:  void vfunc()
{
 cout<<" Der1 clas vfuncs\n";
};  }
```

```
Void main()
{ clescrs;
Base *bptr;
Base bobj;
Derived dobj;
Der1 dobj1;
bptr = & bobj;
bptr -> vfunc();
bptr = & dobj;
bptr -> vfunc();
bptr = & dobj1;
bptr -> vfunc();
getchar;
}
```

O/P: Clo Base clas vfuncs
Derived clas vfuncs
Der1 clas vfuncs

97

→ In Baseclass pointer we can read the objects of Derived class, but a derivedclass pointer can't read Baseclass object. (By typecasting if may possible).

In this all the 3 classes contains 3 virtual function (each class has one virtual function)

```
#include<iostream.h>
      <conio.h>
class Test
{
  public:
      void abc()
        {
        }
};
  void main()
{       classes;
   Test t1;
cout<<"sizeof(t1)<<endl;
 getches;
}
```

O/P: 1 byte.

```
#include<iostream.h>
      <conio.h>
class Test
{
    public:
        virtual void abc()
          {
          }
};
  void main()
{ classes;
   Test t1;
cout<<sizeof(t1)<<endl;
 getch();
}
```

O/P: 2 bytes

The virtual functions will be takesplace at runtime & it also ~~Canbe~~ be invoke just-like a normalmember function with (.) operator. This canbe takesplace by the virtual referencer.

Definition of Overridden:

A function declared as virtual in the Baseclass, and redefined in the derived class is the implementation of the overridden functions.

The prototype for a redefined (or) overridden

Virtual function must exactly match the prototype specified in the baseclass.

prototype encompasses not only signature, but also the return data type of a function.

// Calling a virtual function through a Base class Reference.

Some Previous prog no. 56

```
        ---
        ---
        ---
Void vf(Base &bptr)
{
  bptr.vfunc();
}

  Void main()
{
    clrscr();
    Base bobj;
Derived dobj;
  Der1 dobj1;

  vf(bobj);
  vf(dobj);
  vf(dobj1);
getchar();
}
```

→ ~~The~~ Virtual Attribute is Inherited.

When a virtual function is Inherited, it's Virtual Nature is also Inherited.

This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class. the virtual function can still be overridden.

No matter how many times a virtual function is inherited is remains virtual.

Ex: above prg-

// Virtual Functions are Hierarchical.

When a function is declared as virtual by a baseclass. It may be overridden by a derived class. However the function doesn't have to be overridden.

when a derivedclass fail to overwide a virtual function, then when an object of derivedclass accen that function, the function defined by the baseclass is used.

How virtual function Works :-

```
Class Shape
{
   public: virtual void draw1()
           {
           }
           virtual void draw2()
           {
           }
};
   Class Circle : public Shape
   {
      public: void draw1()
              {
              }
              Void draw2()
              {
              }
};
   Void Main()
   {
      Shape *P, Q;
      Circle C;
      P = &Q;
      P -> draw2();
      P = &C;
      P -> draw2();
   }
```

Address of VTABLE + Address of Slot.

V Table of SHAPE

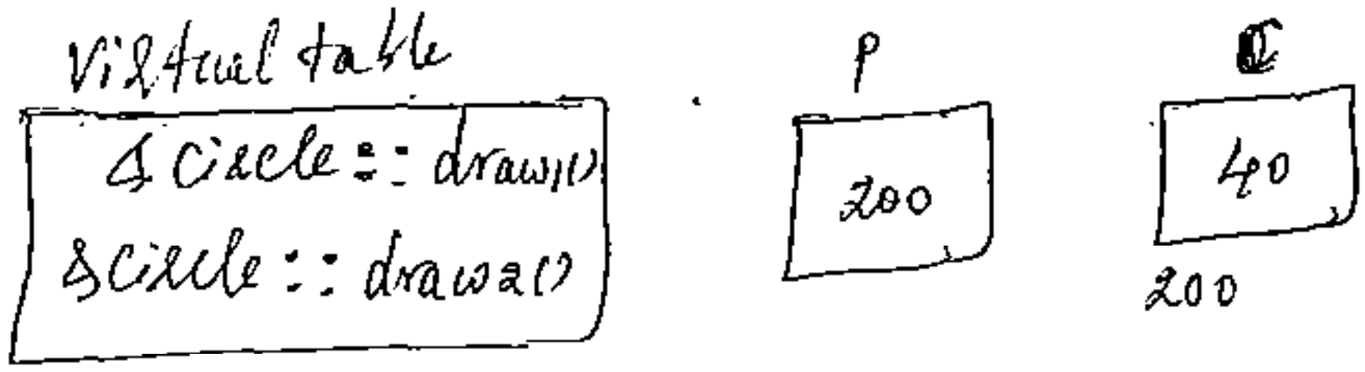
&Shape::draw1()
&Shape::draw2()

60

P: 100

Q: 60

100



V Table is a virtual function table that contains address of virtual functions of draw1() & draw2() of Shap class { } { }
and percles one Vtable is existed

→ Circle class one V table contains draw1() & draw2()
{ } { }

When this VTable is constructed?

Whenever a class is loaded in memory at first time.

The Order is Important of the function Storage.

Order is always depends on Baseclass. In this draw1() is stored in 1st Slot. draw2 is Stored in 2nd Slot.

Order of Address in VTable is always controlled by the order of functions in the Baseclass not in the Derivedclass.

Either in the derivedclass the order is change it will not effect.

When we define p → draw2(); The compiler tells which draw2() function has called, hence it can't bind this function at the early Stage of Compilation.

Late binded i.e dation of draw2() is post-poned during execution.

The stmt- p→ draw2() is expanded into address of vtable + offset of slot.

→ Offset means slot no deided at compilation.

→ Addres deided at execution stage.

→ Slot no is deided wether it is a Vtable (or) circle (or) shape at execution time, by this we are invoking exact functions.

// SHOW ME THE CASES:

// Case1

```
Void main()
{ shape s;
  circle c;
  int a;
  cin >> a;
  if (a==1)
  fun(&s);
else
  fun(&c);
}
void fun(shape *p)
{ p→draw2();
}
```

// case2

```
Void main()
{ shape s1,s2;
  circle c1, c2;
shape *p[] = {&s1, &c1, &s2, &c2};
for(int i=0; i<4; i++)
  fun(p[i]);
}
Void fun(shape *p)
{ p→draw2();
}
```

CH implements a Late binding by setting up a Vtable. The Keywold virtual tells the compiler it should not perform early binding.

Instead, it should automatically install all machanisms necessary to perform late binding. The compiler creates a

Single table called VTable for each class that contains virtual functions.

The Compiler place the address of the virtual functions for that particular class in the VTABLE. A virtual table is therefore an array of virtual functions pointers stored by the compiler as a table called as VTABLE.

Each instance of the class has a pointer to it's class wide VTABLE. Using this, the compiler can achive ~~VTABLE~~ the dynamic binding.

Pure virtual functions

When a virtual function is not redefined by the derived class, the version defined in the base class will be used. However in many situations there can't be any meaningful definition of virtual function within a baseclass.

For example: a base class may not be able to define an object sufficiently to allow a baseclass virtual function to be created.

Further in some situations, you will want to ensure that all derived classes compulsorily override a virtual function.

To handle these two situations. C++ supports pure virtual functions. A pure virtual function is a virtual function that has no ~~that~~ definition in the baseclass.

To declare pure virtual function to use this general form

declaration

↳ virtual type func_name (parameter list) = 0;

When a virtual function is declared pure, any derived class ~~can~~ must provide it's own definition of the virtual function. If the derived class is fail to override the pure virtual function, a compile time error will ensure.

~~The base class number~~

```
#include <iostream.h>
    <conio.h>
class Shape
{
  protected: R, L, B;
  public:
    virtual float Area() = 0;
    virtual float circumference() = 0;
};
class Rectangle : public Shape
{
  public:
      void getLB()
      {
cout << "Enter the Length:";
cin >> L;
cout << "Enter the Breadth:";
cin >> B;
}
    float Area()
    {
      return L * B;
    }
  float circumference()
  {
    2*(L+B);
}
};
```

```
class circle : public Shape
{ public:
    void GetRadium()
    {
cout << "Enter Radium:";
cin >> R;
}
  float Area()
  {
    return 2*3.14
          3.14 * R * R;
  }
 float circumference()
 {
   return 2*3.14*R;
 }
};
Void Calculate (Shape &S)
{
cout << "\n Area:" << S.Area() << endl;
cout << "\n circumference:" << S.
     S.circumference() << endl;
}
```

```
main()
{
  circle C;
  Rectangle R;
 C. Get Radious();
 Calculate (C);
  R. Get-LB();
 Calculate (R);
  getch();
  Return 1;
 }
```

Abstract Classes: A class that contains at least one pure virtual function is said to be abstract. An abstract Class can't be ~~initialized~~ ~~initiated~~ instantiated since it has one (or) more pure virtual functions.

Instead, an abstract constitues an incomplete type that is used as a foundation for derived classes.

Although you can't create Objects of an abstract class. you can create pointers & references to an abstract class.

This allows abtract classes to support Runtime polymorphism which relies upon base class pointers (or) references to select the proper virtual functions.

✓ Constructors & Destructors in Late binding

```
#include <iostream.h>
          <conio.h>
 Class A
{ protected: int i;
```

```
public:  A()
        {
        Cout<<"\n A() Constructor called.";
        }
       ~A()
    {
Cout<<"\n A() destructor called.";
    }
};
Class B: public A
{
   protected: float f;
   public:    B()
              {
   Cout<<"\n B() Constructor called.";
              }
             ~B()
        {
   Cout<<"\n B() Destractor called.";
        }
};
Class C: public B
{
   protected: char name[15];
   public:    C()
              {
    Cout<<"\n C() Constructor called.";
              }
             ~C()
        {
    Coub<<"\n C() Destractor called.";
        }
};
```

```
void main()
{
  A *obj;
 int choice;
 do
 {  clrscr();
 cout<<"\n Call objects of";
 cout<<"\n 1. Class A \n 2. Class B \n 3. Class C \n 0. Exit:";
 cout<<"\n Enter Choice [0-3] :";
 cin >> choice;
 switch(choice)
 {
 case1: obj = new A;
        delete obj; break;
 case 2: obj = new B;
        delete obj; break;
 case 3: obj = new C;
        delete obj; break;
 }
   getch();
 } while( choice >0 && choice<4);
}
```

o/p:

Call objects of
1. Class A
2- Class B
3. Class C
0 - Exit
Enter the choice [0-3]: 1
A constructor called
A Destructor called

Enter the choice [0-3]: 3
A Constructor called
B " "
C " "
A Destructor called.

In this situation proper disposing not taken place
So always use Virtual destructors, rather than normal
destructors.

Using Virtual destructor:

```
class A
{ protected: int i;
  public: A()
        {
  cout<<"In A Constructor called.";
        }
    virtual ~A()
     {
cout<<"In A Destructor called.";
    }
};
void main()
{
. . . .
. . . .
}
```

O/P:
Enter the choice 0-3: 3
A Constructor called
B " "
C " "
C Destructor called
B " "
A " "

// Virtual distructors are possible. Why virtual constructors are not possible.

Constructor is always a functions that is specific to that class only Virtual functions are works with all objects of a class

→ Constructors demands the same name of it's class.

→ Constructors can be overloaded & virtual functions should be overridden.

→ Virtual function object ptr-VF can't directly up on the object but instead it working indirectly to the object

→ Con should fire even along with obj is created.

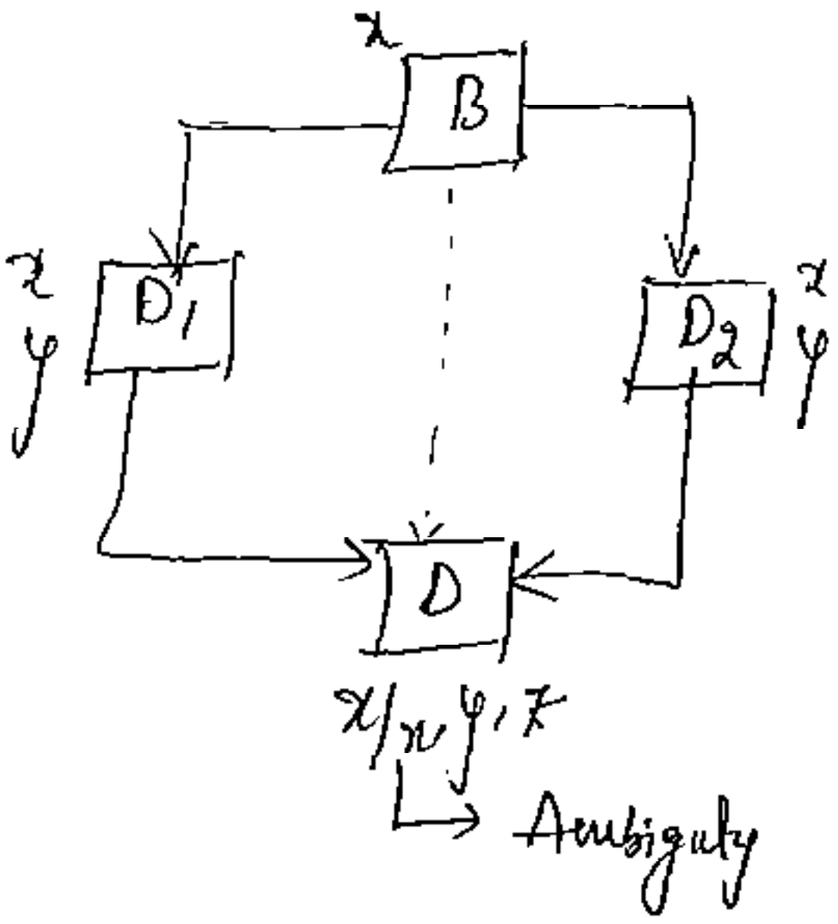
→ Once destructor will work all obj's of the class. is i.e Virtual destructors are possible.

→ Virtual destructors can manage technologically more obj's.

→ Virtual destructors is life's death of virtual program.

Multiple Inheritence

(virtual Inheritence)




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

class A
{ protected:
    int a;
  public: A()
          { a=5; }
};
class B: public A
{ protected:
    int b;
  public: B()
          { b=6; }
};
class C: public A
{ protected:
    int c;
  public: C()
          { c=7; }
};
class D: public B, public C
{ private:
    int d;
  public: D()
          { d=a+b+c; }
    void sum()
    { cout<<"sum:"<<d; }
};
void main()
{ clrscr();
  D d1;
  d1.sum();
  getch();
}
```

If you run this above program, it raises an Error of ambiguity. to overcome this problem to solve in 2 ways.

1. Scope access operator. 2. Virtual keyword.

(1) Class D : public B, public C

{ private: int d;

public: D()

{

d = B::a + b + c; → Scope access operator

}

void Some()

{

}

};

(2) Using Virtual Keyword:

Class A

{ ---

};

Class B : virtual public A

{

};

Class C : public virtual A

{

};

→ Left to Right.

Class D : public B, public C

{

public: D()

{

d = a + b + c;

}

};

Here 'a' property comming from 'B' Class.

EXCEPTION HANLING

Exceptions is Knowthing but Error handling. we can handle runtime error's.

The possible runtime errors are

1. Division by Zero.
2. Sqrt (-ve value)
3. Invalid Subscript for arrays.
4. Illegal memory without allocn.
5. printer without paper

→ Not Everybody can Understand the runtime error ~~and~~ Hence anticipate all the possible runtime errors they may occure in the program & handle them.

→ the ideas is to display user frendly messages Instead of runtime errors.

→ There are 3 reserved in the C++ to handle the Exceptions they are, try, throw, catch.

1. throw :

Syn: throw value;

eg: throw 100;
throw 10.68;
throw 'g';
throw "hyd";
throw obj;

When throw stmt is executed the controle goes to the catch block, Statements following throws are exped.

the next stmt executed after throw is catch.

We must write throw stmts within try block

For every throw there must be a Coresponding catch.

If is suitable catch is not found a runtime error occurs.

// try Block;

try

{ stmt1;

stmt2;

stmt3;

}

→ perform data validation with in tryblock.

→ read data & validate and execute throw stmt if user enter invalid data.

// catch Blocks.

catch (datatype var)

{ stmt 1;

stmt 2;

}

→ display user freudly messages & perform some other actions within catch block.

⟹ A catch block is executed when throw is executed.

→ It is not executed on it's own.

→ although it comes in the flow of control catch is not executed without throw being executed

→ try and catch must exist in the same function.

there can be more than one catch block, but only one among them is executed.

→ No two catches can have same signature.

→ Catch ~~Class~~ can take only one parameter.

→ the parameter contains the value passed from throw.

→ Execution is sequential after catch block execution

→ There can be a default catch.

```
catch(...)
{ stmt1;
  stmt2;
};
```

→ Default catch block must be last block. ~~If is~~

→ It is executed only when Non of the previous catches are not executed.

⇒ Write a program to raise Only One Exception Handling.

```
#include<iostream.h>
#include<conio.h>
main()
{
  int a,b,c;
try
{
cout<<"Enter any 2 values\n";
```

```
cin>> a>> b;
   if( b==0)
   throw " Division by Zero is Not possible"\n";
    c= a/b;
   cout<< c;
} // try
   catch (char *s)
   {
   } cout<<s;
 } // catch
   getche();
   return 0;
} // main.
```

The Disadvantage of Exception in C++ is

1. We can write a program perform data, Validation without try, ~~catch~~ throw, catch.

```
if (b==0)
{ cout << " Division by Zero is Not possible\n";
}
else
c= a/b;
cout<<c;
```

} Thus use if else stmt to perform data validation.

→ There is No predefined Exception in C++ we must use throw stmt explicitly inside to goto catch block.

i.e the control don't goto catch block automatically, When the runtime error occurs.

Example 2:

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

main()
{ int rno, m[3], i, tot;
   float Avg;
   char Sname[30];

  try
  {
   cout<<"Enter roll no:";
    cin>>rno;
    if (rno<1)
     throw 100;
  }
   catch (int i)
   { do
     {
      cout<<"Less than Zero not accepted\n";
   cout<<" Again Enter Correct no:";
    cin>> rno;
      }
     while (rno<1);
   }
  cout<<" Enter the Student name\n";
    cin>> Sname;
  for(i=0; i<2; i++)
  {
   try
   {
     cout<<" Enter the Marks"<<i+1<<":";
      cin>> m[i];
    if((m[i]<1) || (m[i]>100)
     throw 200;
```

```
catch (int i)
{ do
  {
cout<<" Marks must be b/w 0-100 re enter in:";
cout<<"Again enter the marks for same subject:";
 cin>> m[i];
  }
  while(m[i]<0 || m[i]>100);
 } //catch
}//for
     tot= m[0]+m[1]+m[2];
     avg= tot/3.0;
   cout<<" tot:"<<tot<<"Avg:"<<Avg;
     getche;
     return 0;
  }
```

Example 3

Exception Handling in Function.

```
#include<iostream.h>
          <conio.h>
 main()
{ void f1(int);// declaration
  try
  { f1(10);
    f2(20);  } // called
    f3(30);
} // try
catch(int i)
{ cout<<"Integer catch in";
}
Ca
```

```
catch (float f)
{
  cout<<" float catch \n";
}
catch (...)
{
  cout<<" default catch \n";
}
getch();
}
```

O/P: float catch.

```
void f1 (int a) callee.
{
if (a==20) throw 100;
if (a==30) throw 'g';
if (a==10) throw 10.65f;
  throw "hyd";
}
```

→ We can write throw in function & call the function. with in try block.

→ If throw is executed in the called function & the corresponding catch is not found & control comes back to calling function. & searches for catch.

Example 4:

Modifying the above prg

```
main()
{
  void f1(int); // declaration.
  f1(10);
  f1(20);
  f1(30);
}
```

Calls

```
    getch();
    return 0;
}//main

void fi(int a)
{
try
  {
  if(a==20) throw 100;
  if(a==30) throw 'g';
  if(a==10) throw 10.65f;
  throw "hyd";
  }//try
catch(int i)
{
 cout<<" Integer catch\n";
}
catch(float f)
{
  cout<<" float catch\n";
}
catch(...)
{ cout<<" default-catch\n";
}
}
```

O/P: float catch
Integer catch.
default-catch.
default catch.

TEMPLATES

Templete = design

~~#include <iostream.h>~~

~~<conio.h>~~

C++ provides a Generic type of prgramming Such prgramming Once we define a function, it canbe use for all types of procedures.

A Generic function defines a generic set of operations that willbe applied to various types of data.

The type of data that the function will operate that upon is passed to it as a parameter.

Through a generic function, a Single general procedure Canbe applied to a wide range of data.

They are a machanism which make it possible to use one function @ class to handle many diff datatypes.

We can def design a Single class / function which Operates on many datatypes instead of having to create a Separate class / function for each type :-

→ ? When applied with function they are called as "Function templates"

Function Templets :-

Disadvantage of overloaded functions:

→ Time Consuming as the function body is rewriten for diff types

→ The program Consumes more disk space.

→ Error located in one funtion needs Correction in each function body.

Advantages of Templets:

1. It provides a way to code reusability.
2. Inheritence & Composition is ~~the way to~~ provide a way to reuse object code
3. Templets provide a way to reuse source code.
4. They Significantly reduce Source code size & increase code flexibility without reducing the type safety
5. In a function templet a data type can be represented by a name that can stand for any type

Syntax:

```
template< class [template-name] >
[templet-name]
return-type fun-name ([Template-name] val-1,
                      [Template-name] val-2)
{
```

body;
}

⇒ [Templet_name] is known as Template argument-type

→ throughout the definition of the function, Wherever a specific datatype is declared, we substitute the template argument-

```
#include<iostream.h>
        <conio.h>
template <class Htype>
void swap(Htype &a, Htype &b)
 {
  cout<<"Size of a:"<< sizeof(a) <<endl;
        Htype t;
        t=a;
        a=b;
 }      b=t;
  void main()
 { clrscr();
  int a=5, b=10;
  cout<<"Before swaping Inter values:"<<"A:"<<a<<"B:"<<b<<
                                                    endl;
  swap(a,b);
 cout<<"After swaping Inter values:"<<"A:"<<a<<"B:"<<b<<endl;
  getch();
  float x=56.45, y=67.17;
 cout<<"Before swaping float values:"<<"X:"<<x<<"Y:"<<y<<endl;
```

```
    swap(x,y);
cout<<"After swaping float values:"<<"x:"<<x<<"y:"<<y<<endl;
  getch();
    char p='K', q='L';
cout<<"Befor swaping char values:"<<"p:"<<p<<"q:"<<q<<endl;
  swap(p,q);
  cout<<"After  "    charvalues:"<<"p:"<<p<<"q:"<<q<<endl;
    getch();
}
```

CLASS Templates: (STL's) Standard template Libraries.

→ Templates caube extended even to classes.

→ they are generally used for data storage classes (Containers)

→ Building a class Template is similar to a function template

Syntax:

```
template < class |Template_name] >
class <class-name>
  {
     body;
  };
```

→ The Template keyword & <class |Template name]> send a signal specifying that the entire class will be a template.

→ The Template argument will be used in every place in the class specification when there is a reference to the type.

→ Function templates instantiate when a function call is encountered. Classes are instantiated by defining our object using the template arguments.

class-name<type> object; Ex: myclass<int> obj;

Compiler time issues:

→ Compiler doesn't generate any code as yet because it doesn't know as yet what data type, the function will be working with.

→ The code generation takes place when the function is actually called from with in the program.

→ When the compiler sees a function call, it knows the type to use, depends on the arg's & Specific version.

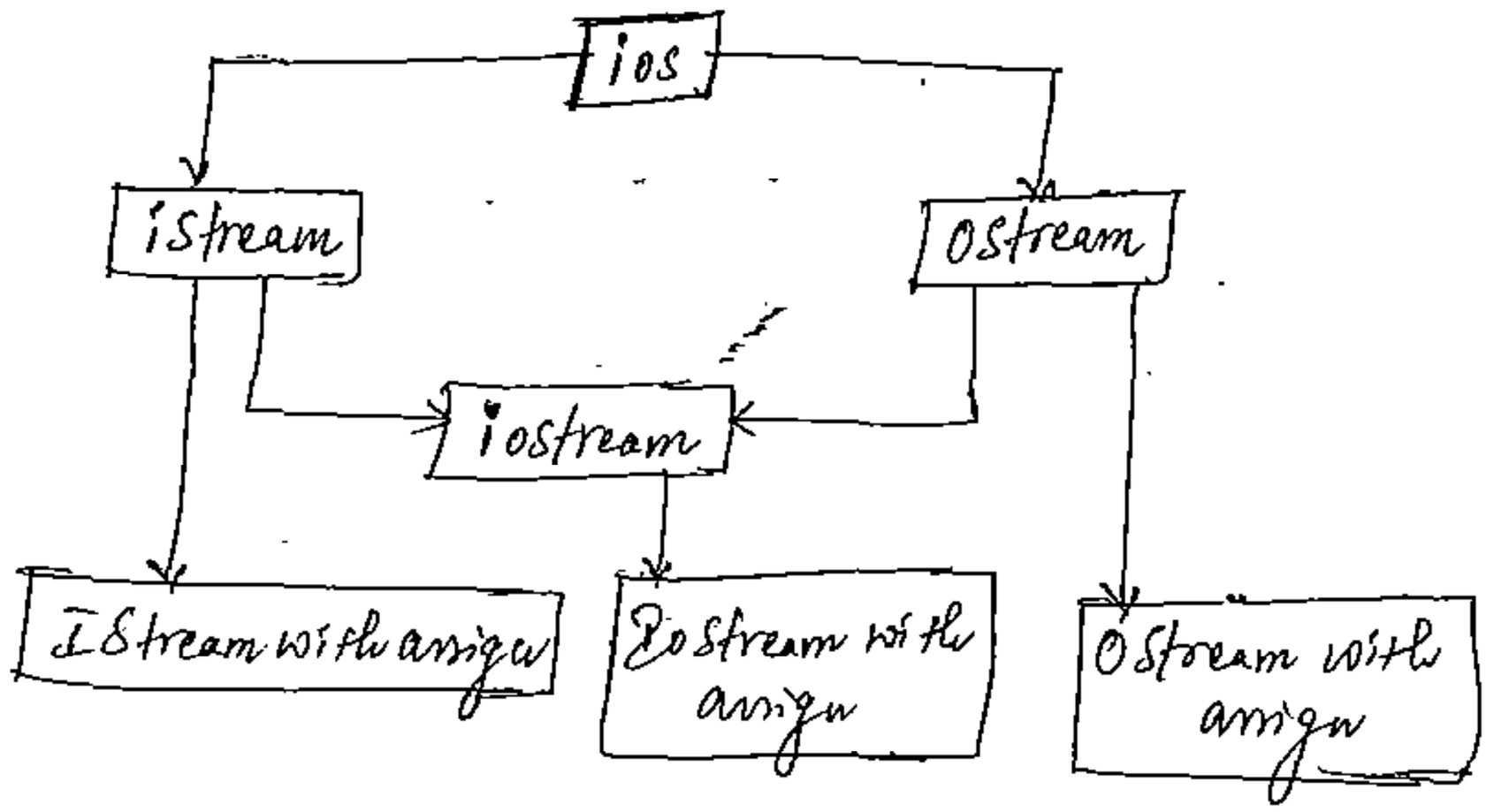
→ The above process is often known as instantiating the function Template.

→ The Compiler generates a call to the newly instantiated function & Inserts it in it's the code.

→ Templates don't save memory, instead they help in not typing the code again & again.

→ The compiler creates them from the generic version that we pass on to it.

Console Management I/O Operations:



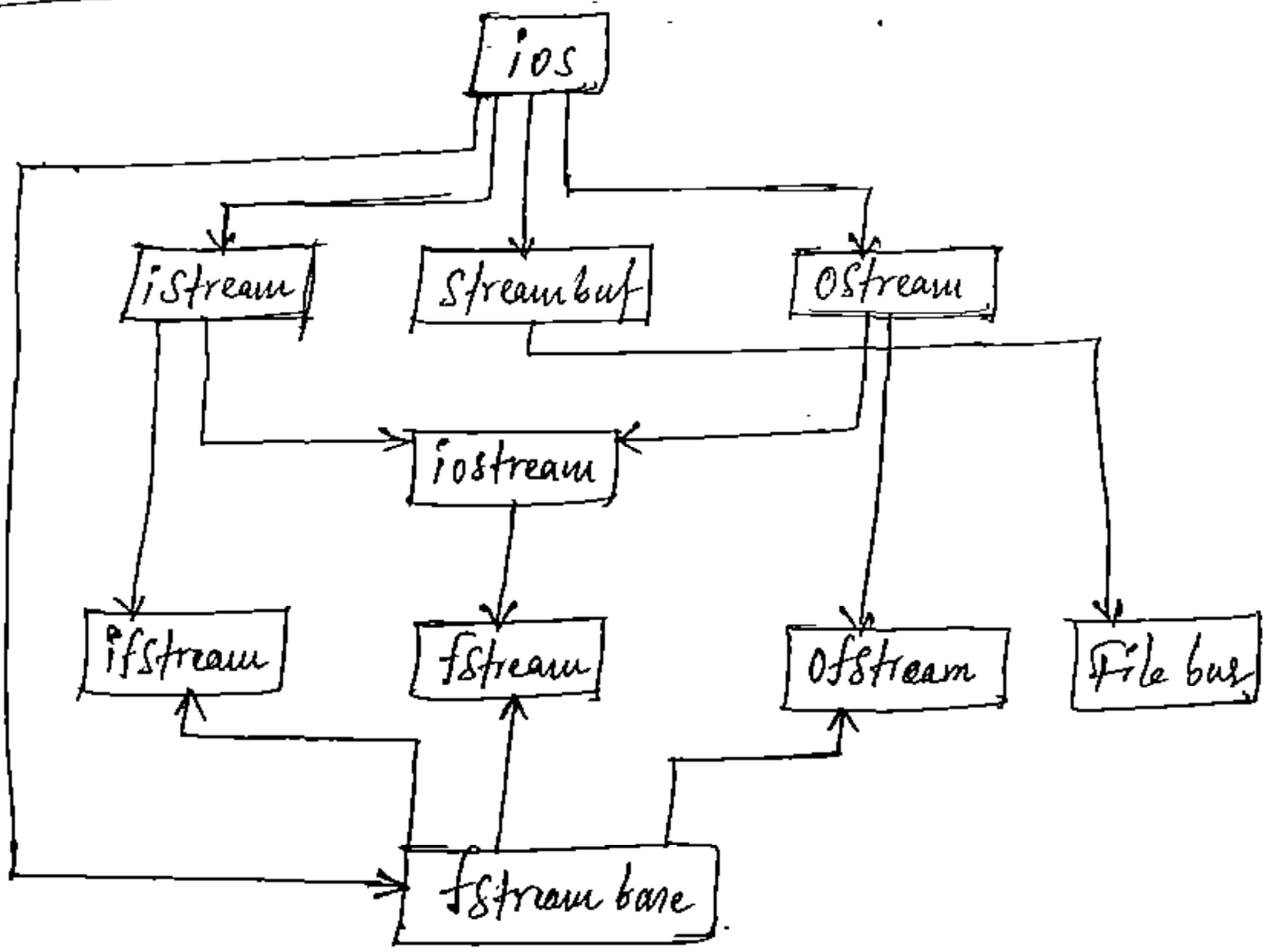
→

#include <iostream.h> → 'put' function display the data after
#include <conio.h> pressing the enter the key.
void main()
{
clrscr();
char ch='F';
cout.put(ch);
cout<<"\n";
cout.put('A');
cout<<"\n";
cout.put(65)<<"\n";
cout<<"Enter the character:";
cin.get(ch);
cout.put(ch);
getch();
}

Example for User Named Manipulators:

#include<iostream.h>
ostream & nextline(ostream &o)
{
return o<<'\n';
}
void main()
{
clrscr();
cout<<"Dont";
cout<<nextline<<nextline;
cout<<"pass";
}

FStream



→ Creating a file using Constructors.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
Void main()
{ clrscr();
  ofstream obj("My first file.txt");
  obj<<"This is my first file\n";
  obj<<"This is a file\n";
  getch();
}
```

// Here ofstream is a class to (predefined.)
Create a file & obj is an object. (user defined.)

→ To set the location of file
go to options In Turbo c editor
↓
options Environment
↳ - variables.
set the output stream
location
c:\-\- ..

→ The diff b/w the cout and inout obj.

→ Cout write the data on Console Stream.

→ Obj writes on the disk.

// Writing and reading from the c++ file-

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
void main()
{
  ofstream out("mfile");
    int i=10;
  char name[]="John";
  out<<i<<endl;
  out<<name<<endl;
  out.close();
ifstream in("mfile");
  int f1;
  char name1[20];
  in>>f1;
  ~~name=fn~~
  in>>name1;
  cout<<"Number:"<<i,<<endl;
  cout<<"Name:"<<name1<<endl;
  getch();
  in.close();
}
```

// write the data onto the file-

Here we can't we any modes because these 2 are default modes.

// Read the data from the file-

→ Create a class class as Employee Store the information on the Disk & Display the information.

```
class Emp
{
  int eno;
  char name[20];
  float sal;
```

```
public:
    void accept()
    {
    cout<<"Enter the no:";
    cin>> eno;
    cout<<"Enter the Name:";
    cin >> ename;
    cout<<"Enter Salary:";
    cin>> sal;
    }
};
void main()
{
    fstream fobj("Employee", ios::out | ios::app | ios::binary);
    emp erec
    char opt;
    do
    {  cout<<"Enter the Employee details:";
       erec.accept();
       fobj.write((char*)& erec, sizeof(erec));
       cout<<"want to continue(Y/N)";
       cin>> opt;
    } while(opt!=n);
    fobj.close();
}
```

Name of the file (↑ "Employee")

mode of append the data on to the file (↑ ios::app)

ios::out ↓ mode: Write the data on the file

ios::binary ↓ for Security.

char opt; — option.

→ Here data stored temporarily in 'erec' obj ~~upto~~ upto ~~class~~ file object ~~of~~ 'fobj' write function execute

→ To implemen the remaing operations like modefy & Delefing always accss in Random accss code which can accss Very fastly.

C++ provides Separate functions for random accss. by using

File pointers.

File pointers:

The C++ I/O system manages two integer values associated with a file.

" GET pointer: It Specifies where in the file, the next I/P (or) read operation will occure.

" put pointer:" It Specifies where in the file next o/p (or) Write operation will occur.

" For every I/P (or) o/p operates that take place, the pointer automatically advances Sequencially.

The "seekg()" and "tellg()" functions allow us to set and Examine the get-pointer.

The "seekp()" and "tellp()" functions perform the actions on the output-pointer.

The above functions allow to accss the file in a non-sequential, random mode.

→ All IOStream classes can be repositioned using either the seekg() (or) seekp() member functions.

→ The functions move the put & the get-pointer to an absolute address within the file (or) a certain no. of bytes from a particular

* The tellg() and tellp() can be used to find out the current-position of the file pointer with in the file

Syntax:

(seekg, tellg, seekp, tellp)

<file-object>. <file pointer> (offset, reference point);

offset is a long integer specifying the no. of byte positions to be moved.

reference point

ios::beg begining of file.
ios::cur Current position of file pointer
ios::end the end of file.

→ The compiler first invokes Base & at the calling of derived class object time also, it invokes the Base class.

Actually it is Not a current function call. because. 1st time Base class object invokes Base ~~calls~~ class function call, 2nd derived class object invokes at that time also Base Class function calls., but actually Derived class function call there,

This is the problem in Compile time polymorphism,
To Overcome that problem we Choose Runtime polymorphism.

Ex: Bank application.

Whenever you choose the other than your bank ATM.
You first communicate with that ATM. but it internally communicates with the your account bank ATM. & updations are coming from there. & you withdraw money from your account the current updated balance will be overside it..

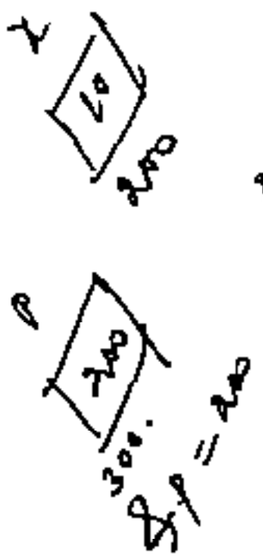
[Ex: overside means replace the old thing with new one]

that way we can use runtime polymorphism in real time.
a It takes bit more time than Compile time polymorphism, but it is more efficient than Compile time polymorphism.

→ ~~File change~~ file → change directly

5 → 0000 0000 0000 0101

5<<2 0000 0000 0000 010100



20

→ This pointer can't WORK's without an Object

This pointer can't work's with the friend function.

Leafer 'this' concept.

'This' can only used with member function.

→ Friend function is used with the Non-member function then Object is Not created for Non-member function. No object, So No possibility of Owning the 'this' pointer

→ ~~If Any Other~~

If our function is a Member of Some other class. then Object is created for that class so in this case 'this' pointer works.

Q. → What is Compile time polymorphism & Runtime polymorphism?

A If is also named as Early binding & Late binding.

Function overloading, Operator overloading are coming under Compile time polymorphism.

→ In Early binding the function caller executes at Compile time

→ In Late binding the function caller executes at Runtime, because we ~~can declare by using~~ that function can be declared as by using the keyword Virtual.

Virtual memory = Runtime allocating memory.

→ In Compile time polymorphism the compiler first- invokes the Base class then Derived class.

& Whenever you create objects for both Base & Derived classes.

→ Predefined operators can't understand the user defined variables.

→ local pointer variables don't know how many elements pointing we have, what type of elements it is.

because local variables stored in Stack area. but Memory allocation takes place in Heap area. ~~that heap area~~ Only that Memory base address given to pointer variable in ~~local~~ stack area.

→ Constructor is also a method. but it is not a method, because constructor is a self intelligence nature ~~or~~ & self initialisation nature.. but method not like this.

→ Domain Knowledge means = Non Technical knowledge Domain = 'concept' (or)

→ blueprint means = it takes all reserved sides.

→ Object is a blueprint of class.

Ex:- ATM cash is an object.
it takes all sides on ATM Machine class
of Banking, withdraws,
Mini statments ... etc

C++ book

Object-oriented Analysis Design
Grady-booch (in Pressman Sir Web side)

→ " " This header file is user define.
< > (or) ' ' predefine

→ Class contains Data members, Main contains object.
~~who~~ how the values stored in ~~to~~ object means,
by call-by-address. Not references
↓ become reference in ambient pointer
Using this.

Note: 'C' & C++ only Supporting Pointers.

C++ is pure object oriented language

by using pointers ~~due to this~~ to find address.

- It is not Secure.

~~due~~ Due to this pointers are removed in Java.

because of security.

→ Because Java used for Developing Internet applications.

http://kilansrinivas.wordpress.com

→ Function Execution always based on 'CPU'

→ Function can call by itself is called Recursion

→ ~~OS Shell → OS S1.c → type S1 to see the program~~

~~typ~~ i

→ exit

→ OS Shell → CPP S1.c → type S1.i to see the program

Library Inclusion. Struct's & our program.

External Scope:

At the time of Our C++ Struct execution the Compiler goto library file & open it & execute the Struct.

for each struct it will go to that same library file. always it jump's to external scope (out of library files).

→ In 'C' language all header files are substitute in the program & our program is occupies large Memory & time Consuming

~~but~~ but- temporary memory not a problem.

→ C++ has only goto useful & predefined Struct to the Library Struct & execute it & & Comeback to main through Inline.

but- inline used for less no. of Structs.

- The parameter ios::app can be used only with the files capable of output.
- Creating a stream using ifstream implies input and creating a stream using ofstream implies output. In these cases it is not necessary to provide the mode parameters.
- The fstream class does not provide a mode by default and therefore, we must provide the mode explicitly when using an object of fstream class.
- The mode can combine tow or more parameters using the bitwise OR operator(symbol |).

Important points to note:-

In C or C++, If the main () function in a program is misspelt, then the error shown would be a linker error but not a compiler error

C and C++ are not completely block structured languages, because a block structured language allows functions within functions.

Multiple Choice Questions:

1. C++ was originally developed by
   1. Clocksin and Mellish 2. Donald E. Knuth
   3. Sir Richard Hadlee 4. Bjarne Stroustrup
2. cfront
   1. is front end of a C compiler
   2. is the pre-processor of a C compiler
   3. translates a C++ code to its equivalent C code
   4. none of the above
3. The following program fragment
```
int i=10;
void main ()
{ int i = 20;
{ int i = 30;
printf("%d, %d", i :: i);
}
}
```
   1. prints 30, 10
   2. prints 30, 20
   3. well result in execution error
   4. none of the above
4. Which of the following is/are procedural languages
   1. Pascal 2. Smalltalk 3. C++ 4. Java
5. A function abc is defined as
```
vid abc (int x = 0, int y = 0)
{ printf("%d, %d", x, y); }
```
   Which of the following function calls is/are illegal? Assume h, g are declared as integers)
   1. abc ( ); 2. abc (h); 3. abc (h, h); 4. none
6. Function prototyping means
   1. checking if the function is declared before its use
   2. checking if the function has a forward reference
   3. checking if the function call conforms to the declaration in type and number
   4. all the above
7. Reusability is a desirable feature of a language as it
   1. decreases the testing time
   2. lowers the maintenance cost
   3. reduces the compilation time
   4. reduces the execution time
8. Choose the correct statements regarding inline functions
   1. it speeds up execution
   2. it slows down execution
   3. it is not used with non-member function
   4. it decreases the code size
9. If many functions have the same name, which of the following information, if present, will be used by the compiler, to invoke the correct function to be used.
   1. The operator :: 2. Object pointer
   3. Function signature 4. All the above.
10. The statements
```
int a = 5;
cout < < "FIRST" < < (a < < 2) < <
"SECOND";
```
    outputs
    1. FIRST52SECOND 2. FIRST20SECOND
    3. SECOND25FIRST 4. An error message
11. Choose the correct remarks
    1. C++ allows the user to define new operators
    2. Some of the existing operators cannot be overloaded
    3. Operator precedence can be changed
    4. All the above

By Kiran Sir

1. An object is declared
2. An object is used
3. A class is declared
4. A class is used

13. Which of the following remarks about the difference between constructors and destructors are correct
1. Constructors can take arguments but destructors cannot
2. Destructors can take arguments but constructors cannot
3. Destructors can be overloaded but constructors cannot be overloaded
4. None of the above

14. The following program fragment
```
void main ()
{ int x = 10;
int &p = x;
printf("%u,%u", &p, &x);
}
```
1. prints 10 and the address of x
2. results in a run time error
3. prints the address of x twice
4. results in compilation error

15. The declaration
int x ; int &p = x;
This remark is
1. true 2. false 3. sometimes true 4. none

16. The following program segment
```
const int m = 10;
int &n = m;
n = 11;
printf("%d, %d", m, n);
```
1. results in compile time error
2. results in run time error
3. prints 11, 11
4. prints 10, 11

17. The following program segment
```
int a = 10;
int &b = a;
a = 11
prints ("%d, %d", a, b);
```
1. results in compile time error
2. results in run time error
3. prints 11, 11
4. none

Choose the correct statements
1. C++ is more strongly typed than C
2. C is more strongly typed than C++
3. size of literal character in C is that of a character 4. none

19. Consider the following program segment static
```
char X[3] = {1234};
cout <<X;
```
A complete C++ program with these two statements
1. prints 1234 2. prints 123
3. prints 1234 followed by some junk
4. will give a compilation error in one of these two statements

20. Choose the correct statements
1. recursive function cannot be declared to be inline
2. recursive function can be declared to be inline
3. two variables having the same scope, cannot have the same name in a C program
4. two variables having the same scope, can have the same name in a C++ program

21. Which of the following operators cannot be overloaded
1. >> 2. ?: 3. Both
4. No such operator exists

The next two questions are based on the following program segment.
```
Class A
{ public :
A(void)
{ cout <<"Howzhat" ;}
~A(void)
{cout <<"Whatizit";}
};
class B : A
{ public :
B(void)
{cout <<"WYSIWYG" ;}
~B(void)
cout << } YACC};
};
```

22. The declaration B x;
1. prints Howzhat WYSIWYG YACC Whatizit
2. prints nothing
3. prints YACC Whatizit Howzhat WYSIWYG
4. none

23. If the main function has the two statements B x; cout <<"done";
the output will be
1. Howzhat WYSIWYG YACC Whatizit done
2. Howzhat WYSIWYG done YACC Whatizit
3. YACC Whatizit Howzhat WYSIWYG done
4. none

24. class Dog : public X, public Y. is an instance of
1. multiple inheritance 2. repeated inheritance
3. linear inheritance 4. none

25. Choose the correct statements
1. a destructor is not inherited
2. a constructor cannot be virtual
3. a destructor can be virtual
4. all the above

26. A function abc is defined as
void abc (int x = 0, int y, int z = 0)
{ printf ("%d, %d, %d, x, y, z);}
Which of the following function calls is/are illegal
(Assume h, g are declared s integers)
1. abc ( ); 2. abc (h);
3. abc (, h) 4. all the above

27. The compiler identifies a virtual function to be pure
1. by the presence of the keyword pure
2. by its location in the program
3. if it is equated to 0 4. none

28. Let class APE be a friend of class SAPIEN. Let class HUMAN is a child class of SAPIEN and let MONKEY be a child class of APE. Then
1. SAPIEN is not a friend of APE
2. APE is not a friend of HUMAN
3. MONKEY isnot a friend of SAPIEN
4. All the above

29. A class having no name
1. is aliowed 2. cannot have a constructor
3. cannot have a destructor 4. all the above

30. If a method is to be an interface between the outside world and a class, it has to be declared
1. private 2. protected
3. public 4. external

The next three questions are based on the following information
int a = 1, b = 2;
a = chg(b);
cout << a << b;

31. If the function chg is coded as int chg (int x)
{ x = 10; return (11);}
then
1. it results in compile-time error
2. it results in run-time error
3. it prints 112 4. it prints 1110

32. If the function chg is coded as
int chg (int &x)
{ x = 10; return (11);}
then
1. it results in compile-time error
2. it results in run-time error
3. it prints 112 4. it prints 1110

33. If the function chg is coded as
int chg (const int &x)
{ x = 10; return (11);}
then
1. it results in compile-time error
2. it results in run-time error
3. it prints 112 4. it prints 1110

34. Choose the correct statement
1. in a struct, the access control is public by default
2. in a struct, the access control is private by default
3. in a class, the access control is public by default
4. in a class, the access control is protected by default

35. Overloading is otherwise called as
1. virtual polymorphism
2. transient polymorphism
3. pseudo polymorphism
4. ad-hoc polymorphism

36. Consider the following program segment
const char *p1 = "To make the bitter butter better "; // stm1
char * const p2 = "Recommend this book 2 others" ; // stm2
p1 = "add some better butter notbitter."; // stm3
p2 = "so that they will get benefitted."; //stm4

*(p1 + 3) = 'A'; //stm5
*(p2 + 3) = 'A'; //stm6
Which of the statement(s) in error
1. stm4 and stm5 2. stm1 and stm2
3. stm1 and stm4 4. stm2 and stm3

37. C++ encourages to structure a software as a collection components that are
1. interactive and loosely coupled
2. not interactive but loosely coupled
3. interactive and tightly coupled
4. not interactive but tightly coupled

38. Which of the following parameter passing mechanism(s) is/are supported by C++, but not by C
1. default value parameters
2. pass by reference
3. pass by constant reference
4. all of the above

39. cout stands for
1. class output 2. character output
3. common output 4. call output

40. The following program
```
void abc (int &p)
{ cout << p; }
void main (void)
{ float m = 1.23; abc(m); cout <<m;}
```
This code
1. results in compilation error
2. results in run-time error
3. prints 11.23 4. prints 11

41. Reference is not same as pointer because
1. a reference can never be null
2. a reference once established cannot be changed
3. a reference doesn't need an explicit dereferencing mechanism
4. all the above

42. If a piece of code can be implemented as a macro or as an inline function, which of the following factors favour implementation as an inline function
1. speed of execution
2. complexity to manipulate as a pointer
3. source code size
4. interacting with other components (like variables in an expression); in the correct way

43. The fields in a structure of a C programare by default
1. protected 2. public 3. private 4. none

44. The fields in a class, of a C++ program are by default
1. protected 2. public
3. private 4. one of the above

The next three questions are based on the following program segment
```
class mho
{public:
mho (void)
{ cout <<"There was";}
mho (Mho &x)
{ cout <<"a certain man; }
mho operator - mho(y)
{ mho ohm; return (ohm); }
};
```

45. If the function main is coded as
mho a, b;
the output will be
1. there was there was 2. nothing
3. a run-time error
4. there was a certain man there was a certain man

46. If the function main is coded as
mho a; a = a - a;
the output will be
1. there was there was 2. nothing
3. a run-time error
4. there was a certain man there was a certain man

47. If the declaration
mho operator - (mho y);
is replaced by
mho operator - (mho &y);
the output will be
1. there was there was
2. nothing
3. a run-time error
4. there was a certain man there was a certain man

The next two questions are based on the following program segment
```
Class A
{ int i1;
protected :
```

```
int i2;
public:
int i3;
};
class B : public A
{ public:
int i4;
};
class C : B
{ };
```

48. The variable i2 is accessible
1. to a public function in class A
2. to a public function in class B
3. to a public function in class C 4. all

49. Which variable(s) is/are accessible from the main function
1. i1 2. i2 3. i3 4. none

50. The following program
```
class abc;
class def
{int i1; // statement 1
protected : int i2; // statement 2
public : int i3; // statement 3
friend abc;
};
class abc
{ public:
void mn (def A)
{cout<<(A.i1=3);cout<<(A.i2=4);
cout<<(A.i3=5)}
};
void main ()
{ defx1; abc x2; x2.mn (x1);}
```
1. will compile successfully if statement 1 is removed
2. will compile successfully if statement 2 is removed
3. will compile successfully if statement 3 is removed
4. will run successfully and print 345

51. Which of the following are good reasons to use a object oriented language
1. you can define your own data types
2. an object oriented program can be taught to correct its own errors
3. it is easier to conceeptulize an object oriented program
4. both 1 and 3

52. When a language has the capability to produce new data types, it is called
1. extensible 2. overloaded
3. encapsulated 4. reprehensible

53. Dividing a program into functions
1. is the key to object oriented programming
2. makes the program easier to conceptualize
3. makes the program run faster
4. both 2 and 3

54. In C++, a function contained within a class is called
1. a member function 2. an operator
3. a class function 4. a method

55. A normal C++ operator that acts in special ways on newly defined data types is called
1. classified 2. overloaded
3. encapsulated 4. glorified

56. An expression
1. usually evaluates to numerical value
2. may be part of a statement
3. both 1 and 2 above
4. always occurs outside a function

57. A relational operator
1. compares two operands and yield a boolean result
2. assigns one operand to another
3. logically combines two operands
4. all of the above

58. A variable defined within a block is visible
1. throughout the function
2. from the point of definition onward in the block
3. from the point of onward in the function
4. from the point of definition onward in the program

59. In C language, the && and || operators
1. compare two numeric values
2. combine two numeric values
3. compare two boolean values
4. combine two boolean values

60. In C language, the break statement causes an exit
1. from all loops and switches
2. only from the inner most switch
3. from the innermost loop or switch
4. only from the innermost loop

61. In C++, the go to statement causes control to go to
1. a function 2. a label
3. a variable 4. an operator

62. In a far loop with a multistatement loop semicolons should appear following
1. the test expression
2. each statement within the loop body
3. the closing brace in a multistatement loop body
4. both 1 and 2

63. In C++, a structure brings together a group of
1. related data items, variables
2. integers with user-defined names
3. items of the same data type
4. all of the above

64. In C++, when accessing a structure member, the identifier to the left of the dot operator is the name of
1. the keyword struct 2. a structure variable
3. a structure tag 4. a structure member

65. An enumerated data type brings together a group of
1. constant values, integers with user defined names
2. related data types
3. items of different data types
4. all of the above

66. A function argument is
1. a varible in the function that receives a value from the calling program
2. a value sent to function by the calling program
3. a value returned by the function to the calling program
4. a way that functions resist accepting the calling programs values.

67. In C++, which of the following can legitimately be passed to a function
1. a structure 2. a variable
3. a constant 4. all of the above

68. How many values can be returned from a function
1. 1 2. 2 3. 4 4. any number

69. In C++, when an argument is passed by reference
1. the function cannot access the arguments value
2. a temproary variable is created in the calling program to hold the argument's value
3. a variable is created in the function to hold the arguments value
4. the function accesses the arguments orginal value in the calling program.

70. Overloaded functions in C++
1. are a group of functions with the same name
2. all have the same number and type of arguments
3. may fall unexpectedly due to stress
4. all of the above

71. In C++ functiosn, a default argument has a value that
1. may be supplied by the calling program or the function
2. may have a constant value
3. may have a variable value
4. none of the above

72. A static automatic variable (in C++) is used to
1. conserve memory when a function is not executing
2. make a variable visible to several functions
3. make a variable visible to only one function and retail a value when a functoin is not executing.
4. make a variable visible to may functions when it is executing and retain a value when a function is not executing.

73. In a class specifier (in C++) data or functions designed private accessible
1. only to public members of the class
2. to member functions of that class
3. only if the passward is known
4. to any function in the program

74. In C++, the dot operator (or class member access operator) connects the following two entities
1. a class and a member of that class
2. a class object and a class
3. a class member and a class object
4. a class objeect and a member of that class

1. Example for function template

```
# include<iostream.h>
# include<conio.h>
template < class type >
void message( char *msg, type f)
{
gotoxy(1,23);
delline();
textcolor(RED);
cout<<msg<<" ! "<<f;
gotoxy(10,24);
textcolor(GREEN);
cprintf("Press any key to Continue.....");
getch();
textcolor(WHITE);
delline();
gotoxy(1,23);
delline();
}

void main()
{
int i;
float f;
char str[10];
clrscr();
cout<<"\n Enter Any Integer :";
cin>>i;
cout<<"\n Enter Any Float :";
cin>>f;
cout<<"\n Enter Your Good Name :";
cin>>str;
cout<<"\n Calling the message with Integer
:";
message(str,i);
gotoxy(1,19);
cout<<"\n Calling the message with float
:";
message(str,f);
}
```

color cprintf()

3. Explicitly overloading function template.

```
# include<iostream.h>
# include<conio.h>
# include<string.h>
# define N 2
class emp
{
private:
int no;
char name[20];
float sal;
public:
void accept( );
void show( );
int sno( );
float ssalary( );
char *sname( );
};

void emp :: accept( )
{
cout << "\n employee no :" ;
cin>>no;
cout << "\n employee name :" ;
cin>>name;
cout << "\n employee salary :" ;
cin>>sal;
return ;
}

void emp :: show ( )
{
cout << "\n displaying the data :";
cout << "\n no :" <<no;
cout << "\n name :" <<name;
cout << "\n salary :" <<sal;
return ;
}

int emp :: sno( )
{
return no;
}

float emp :: ssalary( )
{
return sal;
}

char * emp :: sname( )
{
return name;
}

int findrecord (emp e[ ], char ch[ ] )
{
```

2. Templates with Constructors.

```
#include<iostream.h>
#include<conio.h>
template <class T>
class sample
{
private:
        T value;
public:
        sample( T n)  : value (n) { };
//constructor
        ~sample () { } //destructor
        void display() {
        cout<<"Contents of the value
="<<value<<endl;
}
};
void main()
{
sample<int>obj1(10);
cout<<"Integers :"<<endl;
obj1.display();
sample<float> obj2(-22.12345);
cout<<"Floating Point number :"<<endl;
obj2.display();
}
```

→ Non member function.

```
for (int i=0; i<N; i++)
if (strcmp (e[i].sname( ), ch)==0)
return i;
return-1;
}

void list (emp e[ ])
{
for (int i=0; i<N; i++)
{
cout << "\n employee :" <<i+1;
e[i].show( );
}
return;
}

template <class type>
int findrecord (emp e[ ], type n, int choice)
{
for (int i=0; I<N; I++)
{
if (choice ==1)
{
if (e[I]. sno( )==n)
 return I;
 }
 if (choice==2)
 {
 if (e[i]. sname( )==n)
 return I;
   }
        }
        re
        }

        Int
        {
        int choice; clrscr( );
        cout << "\n query by.";
        cout << "\n 1.employee no :";
        cout << "\n 2. employee name :";
        cout << "\n 3. employee salary :";
        cout << "\n 0.exit";
        cout << "\n enter your choice [0-3]
   :";
        cin>>choice;
        return choice;
        }

        void main( )
        {
        emp e[N];
        cout << "\n enter the employee
details :";
        for (int I=0; I<N; I++)
        {
        clrscr( );
        cout << "\n employee :" <<I+1;
        e[i].accept( );
        }
        cout << "\n displaying the data";
        list(e);
        getch( );
        int choice ;
        do
        {
        int k = -1;
        choice = menu();
        switch (choice)
        {
        case 1: int n; cout << "\n enter the
employee no:" ;
        cin>>n;
        k=findrecord (e, n, 1); break;

        case 2 : char ch[15];
        cout << "\n enter the name :";
cin>>ch;
        k=findrecord (e, ch); break;

        case 3 : float f;
        cout << "\n enter the salary :";
cin>>f;
        k=findrecord (e, f, 2); break;
        }
        if (k>=0)
        e[k].show( );
        getch();
        } while(choice>=0&&choice<4);
        }
```

```
#include <iostream.h>
#include <conio.h>
#define N 5
template <class type>
class array //  Class Template
{
protected :
type a[N];
public :
void show( );
void get( );
};



template <class type>
   void array <type>:: get( )
   {
   for (int i=0; i<N; I++)
   {
   cout << "\n enter values for a ["<<i<<"]
:";
   cin>>a[i];
   }
   return;
   }
```

```
template <class type>
void array<type> :: show( )
{
cout.setf(ios:: showpoint);
type sum = 0;
for (int i=0; i<N; i++)
{
cout << "\n a ["<<i<<"] :" <<a[i];
sum = sum+a[i];
}
cout << "\n the sum of elements is :"
<<sum;
getch( );
return;
}

template <class type>
class newarray : public array <type>
{
Inheritance
public :
void get( );
void show( );
};

template <class type>
void newarray <type> :: get( )
{
cout << "\n Enter your goes name
for (int i=0; i<N; i++)
cin>>a[i];
getch( );
return;
}

template <class type>
void newarray <type> :: show( )
{
for (int i=0; i<N; i++)
cout<<a[i];
getch( );
return;
}

void main( )
{
int choice;
do
{
clrscr( );
cout << "\n Illustration of class templates
\n\n Menu \n";
cout << "\n 1. integer type array \n 2. float
type array \n 3. string \n 0. exit";
cout << "\n enter choice [0-3] :";
cin>>choice;
switch (choice)
{
case 1: array <int> obj;
cout << "\n enter integer values :";
obj.get( );
cout << "\n displaying the values :";
obj.show( );
break;
case 2: array <float> obj1;
cout << "\n enter float values :";
obj1. get( );
cout << "\n displaying the values :";
obj1.show( );
break;
case 3: newarray <char> obj2;
obj2. get( );
obj2. show( );
break;
}
} while(choice <0 && choice<4);
}
```

// Template Inheritance

```
#include<iostream.h>
#include<conio.h>
template <class T>
class sample
{
private:
    T value,value1,value2;
public:
    void getdata();
    void sum();
};
template <class T>
void sample <T> :: getdata()
{
cin>>value1>>value2;
}
template <class T>
void sample <T> :: sum()
{
T value ;
value = value1 + value2;
cout<<"Sum :"<<value<<endl;
}
void main()
{
sample<int>obj1;
sample<float>obj2;
cout<<"Enter any two integers :"<<endl;
obj1.getdata();
obj1.sum();
cout<<"Enter any two floating point
numbers :"<<endl;
obj2.getdata();
obj2.sum();
}
```

Defining the function out of class. Use the follow syntax.

→ Templates can be defined for constructors also.

→ When you define class has Template, member functions also by default- template.

→ Secondary memory devices, Harddisk, Ploppy disk.