

1. Principles of object oriented programming.

* procedure oriented programming :-

- In POP language focus is on doing things.
- procedure means collection of steps required for solving any problem.
- 'C', 'COBOL', 'FORTRAN' and 'PASCAL' are procedure oriented languages.
- In POP large programs are divided into smaller programs known as function.
- A function is called as procedure.
- POP languages are used Top-Down design approach.
- Data moves openly from one function to another function.
- Data is not hidden & functions are share global data.

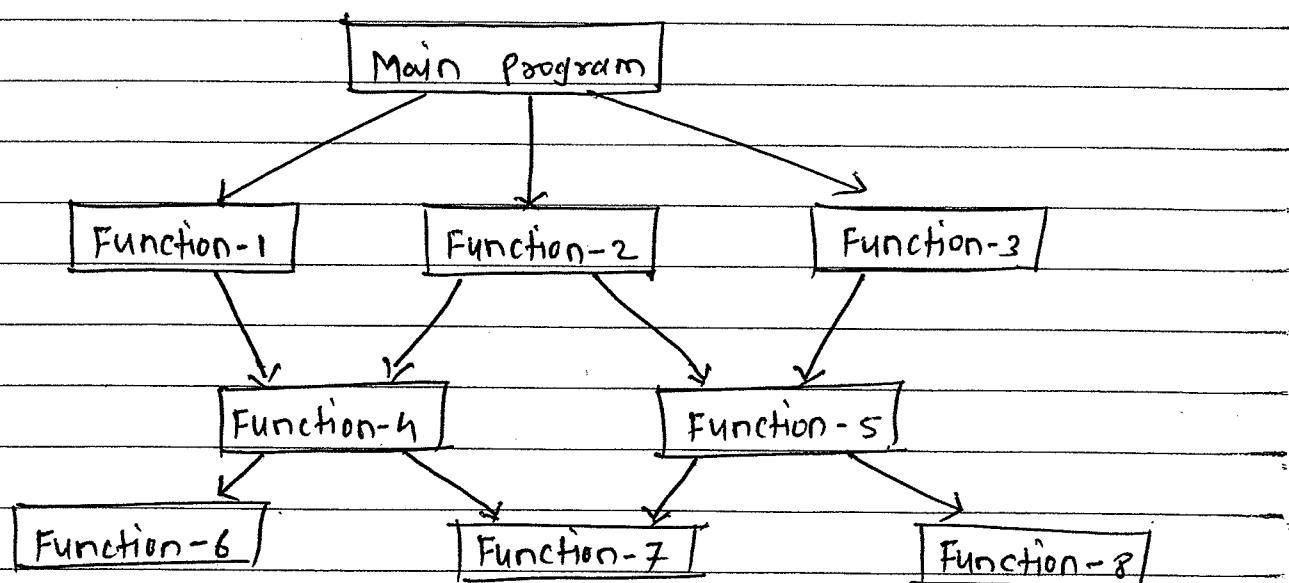


fig :- Structure of procedure oriented program

- In above figure, we can see large programs are divided into multiple small programs (functions).

characteristics of P.P :-

- 1) Emphasis is on doing things.
- 2) Large programs are divided into smaller programs.
- 3) Most of the functions share global data.
- 4) Data moves openly from one function to another function.
- 5) Top-Down approach used in programs design.
- 6) Data in P.P is not hidden.
- 7) Modular approach programming language.

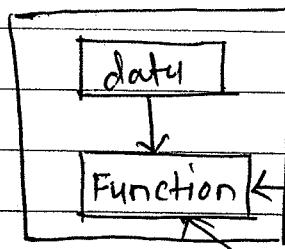
Disadvantages of P.P :-

- 1) It does not solve real world problems very well.
- 2) No security for data

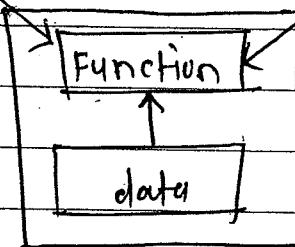
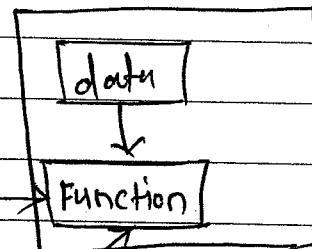
* Object oriented programming language :

- In OOP emphasis is on data rather than procedure.
- In OOP data is hidden & cannot be accessed by external functions.
- New data & functions can be easily added.
- Bottom-up approach used to design programs.
- Data cannot be moved openly from one function to another function.
- classes, inheritance, function overloading all these new concepts are added.
- objects are basic run time entities.
- In OOP everything is done by using objects.
- object is a collection of data & function.

object - 1



object - 2



object - 3

fig: structure of data & functions in oop:

- function of one object can communicate to function of another object.

eg.

simula

C++, Java, SmallTalk, etc} are OOP languages.
Object Pascal, Eiffel }

Characteristics of OOP:

- 1) Bottom-up approach used for programs design.
- 2) Emphasis is on data rather than procedure.
- 3) data & functions are present in object.
- 4) Data is hidden & cannot accessed by external fun.
- 5) objects can communicate to each other through functions.
- 6) New data & functions are easily added.
- 7) programs are divided into objects.
- 8) provide security for data.

Advantages of OOP:

- i) It shows reusability feature because of inheritance. It save time in development.
- ii) Data is hidden because of this programmers develop secure programs.
- iii) we can create multiple object without any problem.
- iv) objects can communicate with each other through functions.
- v) object oriented system can easily upgraded from small to large systems.
- vi) software complexity can be easily managed
- vii) It is very simple, small & easy to understand.

* Applications of OOP:

- 1) Real time systems
- 2) simulation and coding modeling.
- 3) object oriented databases
- 4) Hypertext, hypermedia and expertext.
- 5) AI & expert systems
- 6) Decision support & office automation system.
- 7) parallel programming
- 8) CAM/CAD systems.

* Difference between OOP & POP:

POP	OOP
① POP stands for procedure oriented programming.	① OOP stands for object oriented programming.
② Emphasis is given on data doing things.	② Emphasis is on data rather than procedure.
③ Large programs are divided into small programs.	③ programs are divided into objects.
④ Data moves openly from one fun ⁿ to another fun ⁿ .	④ Data is hidden so it cannot moves openly.
⑤ functions share global data. Data is not hidden.	⑤ Data is hidden & cannot be accessed by other functions.

POP

⑥ TOP-down design approach

⑦ difficult to add new data & functions

⑧ It does not solve real world problem very well

⑨ eg.

C, FORTRAN, PASCAL

OOP

⑥ Bottom-up design approach

⑦ easy to add new data & function.

⑧ It is used to solve real life problem.

⑨ eg.

C++, JAVA

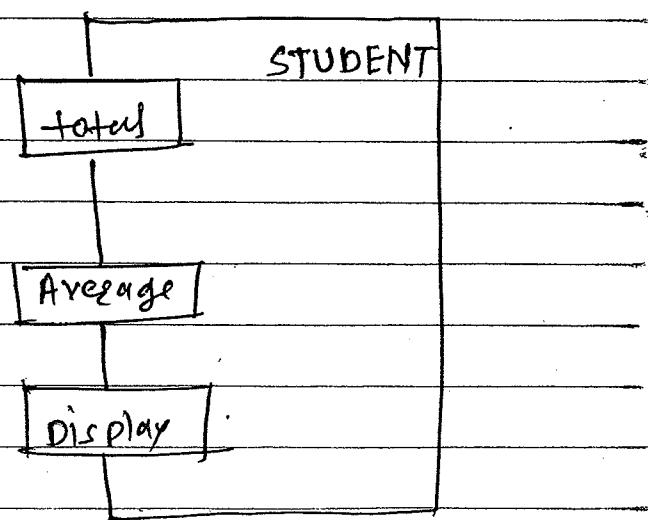
* Basic concept of object oriented programming:

- There are eight basic concept of oop:
- 1) objects
- 2) classes
- 3) Data abstraction
- 4) Data encapsulation
- 5) Inheritance
- 6) polymorphism
- 7) Dynamic Binding
- 8) Message passing.

1) objects:

- objects are basic run time entities.
- they may represent place, bank account, table of data or any data that the program must handle.
- Each objects contains data & functions.
- Objects are created from classes.
- Objects are communicate to each other by passing messages.
- following diagram shows representation of objects.

Object : STUDENT
DATA: Name
DOB
Marks
FONCTIONS:
total
Average
Display



eg.

mango, apple are objects of fruit class.

2) classes:

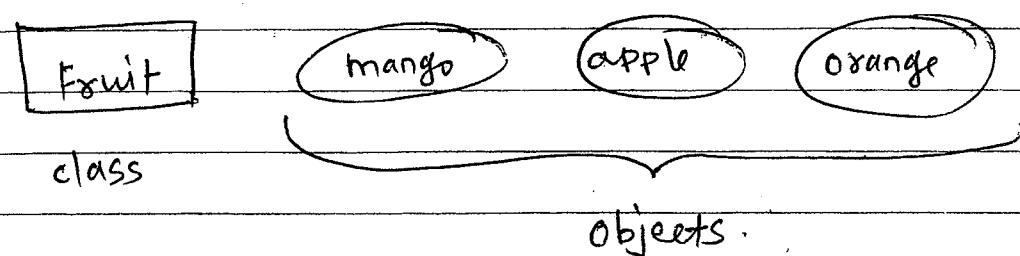
- class is a collection of objects of similar type.
- class is a group of objects of similar properties.
- class contains data & function.
- we can create multiple objects from class.
- classes are user defined data types.
- classes show data abstraction & data encapsulation properties.
- class is abstract data type (ADT).
- eg:

mango, apple and orange are objects of fruit class.

Syntax:

className objectName;

eg. Fruit mango, apple, orange;



③ Data abstraction :

- Data abstraction is one of the basic concept of OOP.
- Data abstraction means to show essential details to the user & hide unwanted details from user.
- classes uses the concept of data abstraction so it is known as Abstract Data Types (ADT).
- classes shows only behaviour of class without showing coding details.
- Data abstraction means to represent only essential details without including background details.

④ Data Encapsulation :

- The wrapping up of data & functions into a single unit is known as data encapsulation.
- It is very important features of class.
- The data of class can not be accessible outside the class.
- only those functions access that data which are declared inside the class.
- functions provides interface b/w objects of program.
- Encapsulation is a mechanism of that keeps data & function safe insi from outside the class.

⑤ Inheritance:

- Inheritance is a process of creating new class by using the concept of old class.
- Objects of one class can acquire the properties of another class is known as inheritance.
- we can define new derived class by using the concept of base class.
- OOP shows reusability property because of inheritance.
-

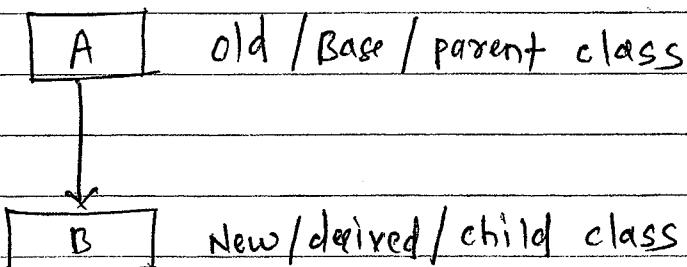


fig: Inheritance

- Above fig. shows class B is derived from base class A
- following are the types of inheritance
 - 1) single inheritance
 - 2) multilevel inheritance
 - 3) multiple inheritance
 - 4) Hybrid inheritance
 - 5) Hierarchical -II-
- Inheritance helps to reduce the size of programs and it saves development time.

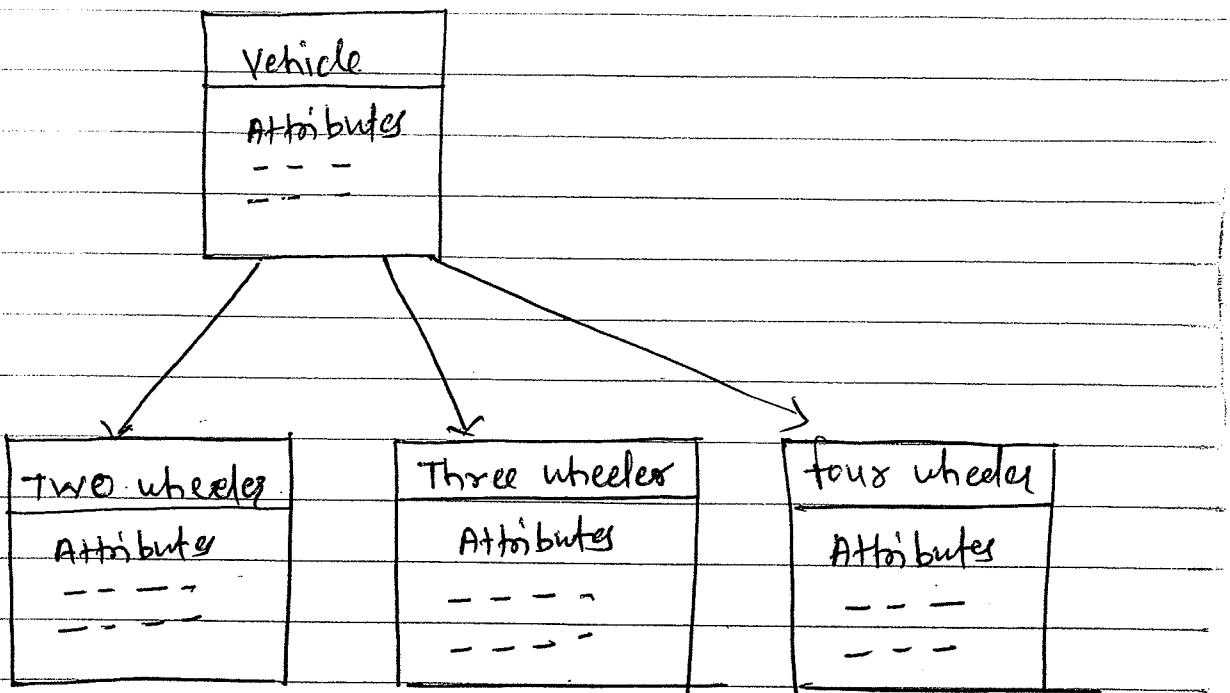


fig: Inheritance property

6) polymorphism :

- polymorphism is another important oop concept.
- polymorphism means ability to take more than one form.
- In polymorphism - poly means 'many' and morphism means 'forms'.
- polymorphism is a greek word.
- single concept = many meanings is known as polymorphism.
- polymorphism supports below two important concepts.
 - ① function overloading.
 - ② operator overloading.
- function name are same but it's arguments

are different is known as function overloading.

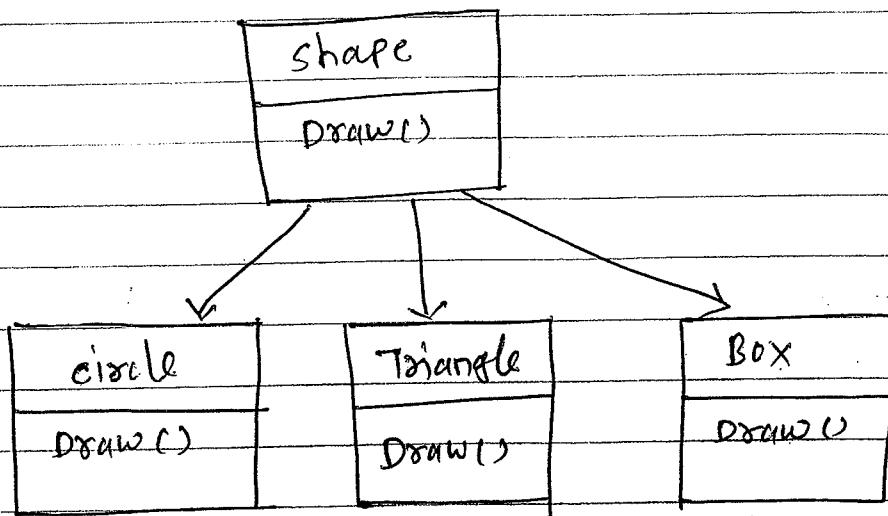


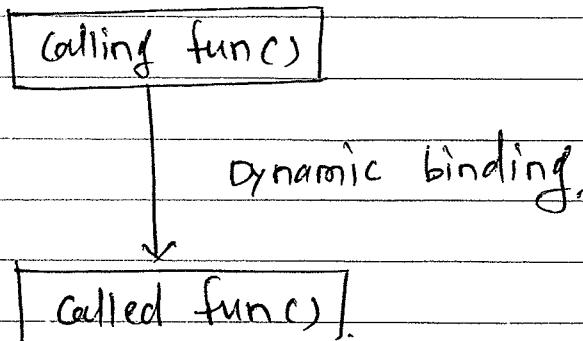
fig: polymorphism:

- In above figure to draw shape shape of circle, triangle and Box, we are using Draw() method.
- Draw() method name are same but it's arguments are different for circle, Triangle, Box.

7) Dynamic binding:

- Binding means linking between calling funⁿ f called funⁿ.
- But this linking not known until the ~~time~~ execution of program. is known as dynamic binding.
- OOP concept polymorphism and inheritance are associated with dynamic bindings.

- Dynamic binding also called as late binding.



⑧ Message passing:

- An object plays very important role in object oriented program.
- objects communicate to each other by passing messages.
- The message passing process involves the following basic steps.

- I) creating classes that define objects
- II) creating objects from class
- III) Establish communication b/w objects.

- objects communicate with each other by sending & receiving information.
- msg passing involves name of object, name of function & information to be sent.

employee • Company (Company-name);

↑ ↑ ↑
object function message/information

* History of C++

- C++ is a new version of c language.
- C++ language was developed by Bjarne Stroustrup.
- C++ language developed at "AT & T bell lab" USA.
- C++ language developed in development starting in 1979.
- Initially it was called "c with classes" but it was renamed C++ in 1983.

* Tokens:

- The smallest individual units in a program are known as tokens.
- C++ programs are written by using tokens
- C++ has following tokens:
 - ① keywords
 - ② Identifiers
 - ③ Constants
 - ④ strings
 - ⑤ operators.

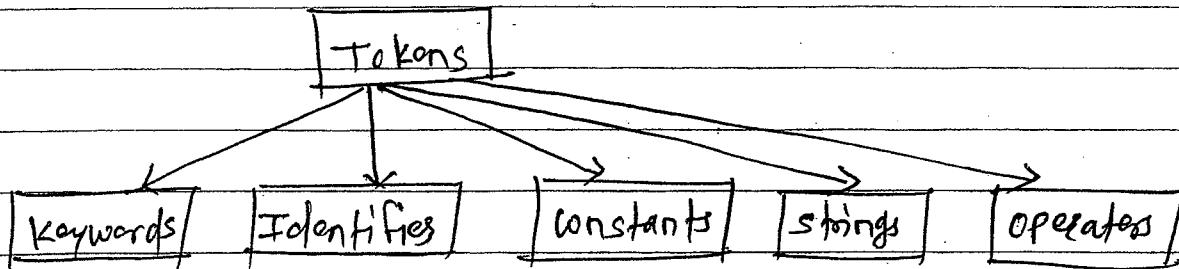


fig: tokens

* Keywords :

- Keywords are reserved words of c++ language.
- Keywords are predefined words.
- Keywords are cannot be used as variable name.
- There are 48 keywords present in c++.

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

fig: c++ keywords

* Identifiers:-

- An identifiers is any name of variables, function, classes etc.
- following rules are used to defining identifiers
 - 1) An identifiers must be unique in program

- 2) It must be start from alphabets or ~~underline~~ underscore (-).
 - 3) keywords cannot be used as identifier.
 - 4) It does not allowed any special symbol except underscore (-).
 - 5) It may contain digits (0-9).
 - 6) It does not allowed any white space.
- eg.

marks, stud_name, no1, no2 // valid
 2011\$no, 123, stud marks // invalid

* Constants :

- constant is a variable which does not change value during execution of program.
- constant variable always written in uppercase letter because it can easily identify.
- There are two diffn ways to define constants in C++:

- ① using const keyword
- ② using #define preprocessor.

① Using const Keyword:

- const is a predefined keyword. It is used to declare constant to variable.

Syntax:

const	datatype	variableName	=	value	
-------	----------	--------------	---	-------	--

Eg.

const int MAX = 10;

Program:

```
#define KioskRate 10
```

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

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

```
Void main()
```

```
{
```

```
const int MAX = 50;
```

```
clrscr();
```

```
cout << "Value of constant variable MAX = " << MAX;
```

```
getch();
```

```
}
```

O/P

Value of constant variable MAX = 50

(2)

Using #define preprocessor:

- #define is a preprocessor directive.

- It is used to declare constant variable.

Syntax:

```
#define VariableName Value
```

Eg.

```
#define max 50
```

program!

```
#include <iostream.h>
#include <conio.h>
#define MAX 50
Void main()
{
    clrscr();
    cout << "Value of const variable MAX = " << MAX;
    getch();
}
```

O/P

Value of const variable MAX = 50

* Basic Data Types :

- Data types are used to declare variables in program.

- Data types decide type of variables & size of variables.

- Syntax:

datatype variableName;

- ex:

```
int a;
float b;
```

C++ data types

User defined type

- structure
- union
- class
- enumeration

Built in type

Derived type

- array
- function
- Pointer

Integral type

Void

Floating type

int

char

float

double

fig! C++ data types

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

* operators in C++

- C++ has a rich set of operators.
- All C operators are valid in C++.
- C++ have some other new operators.

Operators	Meanings
<<	Insertion operator
>>	Extraction operator
::	Scope resolution operator
::*	Pointer to member declaration
→*	Pointer to member operator
.*	pointer to member operator
delete	Memory release operator
endl	Line feed operator like 'ln'
new	Memory allocation operator
setw	Field width operator.

* Scope Resolution Operator

- scope resolution operator is denoted by pair of colons (::).
- If local variable and global variable has same name then treat variable only local variable live in main method because local.
- If you are going to print that variable then local variable value will be printed on output screen.

- In this case how to print or access global variable.
- Above problem can be solved by using scope resolution operator (`::`)
- Scope resolution operator is used to access global variable.

Syntax:

`::VariableName`

Program:

```
#include <iostream.h>
#include <conio.h>
int a=10;
Void main()
{
    int a=15;
    clrscr();
    cout << " Local a = " << a;
    cout << " It Global a = " << ::a;
    ::a=20;
    cout << "\nLocal a = " << a;
    cout << " It Global a " << ::a;
```

`getch();`

`}`

`Output`

Local a = 15 Global a = 10

Local a = 15 Global a = 20

* Memory Management Operators:

- In 'C', we can allocate memory dynamically by using malloc() & calloc() functions.
- C++ provides new operator for dynamic memory allocation.
- delete() function used to delete memory which is allocated by using new operator.
- object can be created by using new operator & destroyed by using delete operator.

Syntax:

```
datatype *ptrVariableName = new datatype;
```

Eg:

```
- char *p;  
p = new char[5]; // reserve 5 bytes
```

```
- int *p;  
p = new int; // allocates 2 bytes.
```

delete operator deallocated memory allocated by new.

Syntax:

```
delete ptrVariableName;
```

Eg:

```
delete a; // delete normal variable memory
```

delete

```
delete []p; // delete complete array memory
```

Program:

#define

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

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

```
Void main()
```

```
{
```

```
char *x;
```

```
x = new char[10]; // allocates memory using new
```

```
strcpy(x, "Computer");
```

```
cout<<"\n string is : "<<x;
```

```
delete x; // delete memory.
```

```
getchar();
```

```
}
```

O/P	
string is: Computer	

* structure of C++ program

- program is a collection of instruction.
- C++ programs contain four sections.

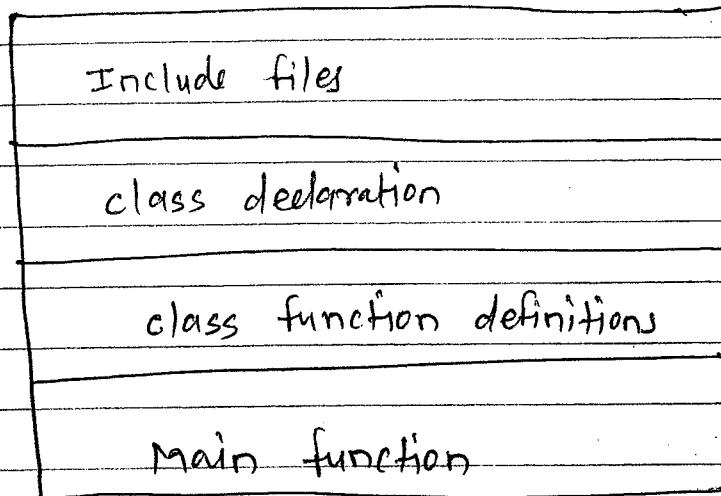


fig: structure of C++ program

- C++ statement terminates with semicolon.
- C++ program is organized in three separate files.
- class declaration is placed in header file.
- definition of member functions go into another file.
- Main program that uses class file is placed in third file which includes previous two files.
- The main function is executed first in C++ program.
- program - simple C++ program

// program to print string → this is comment

// program to display string → This is comment

```
#include<iostream.h> } → Header files  
#include <conio.h> {  
void main() → main function.  
{  
    clrscr();
```

cout << "welcome to world of c++"; → // C++ output

```
getch();  
}
```

O/P

welcome to world of c++

* Input operator/stream :

- It is used to read input from user through keyboard.
- cin is a predefined object and >> is a extraction operator both are used to read value from user.
- syntax:

```
    cin >> variableName;
```

eg:

```
    cin >> n01;
```

- In above example program controller read value from user and stored int in not variable

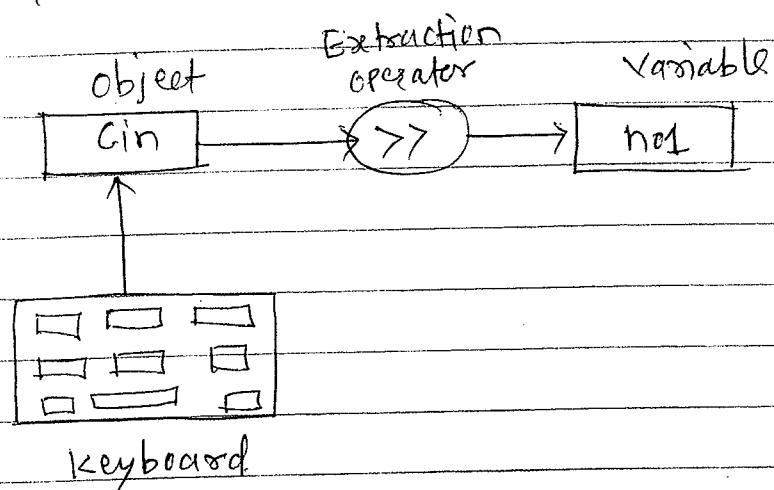


fig: Input with cin operator

Eg.

1) `int a;` 3) `char c;`
`cin>>a;` `cin>>c;`

2) `float b;`
`cin>>b;`

- we can get more than one I/O using cin object

syntax:

`cin>>x1>>x2>>x3---;`

Eg.

`int a, b, c;`
`cin>>a>>b>>c;`

* Output operators / stream

- It is used to display output values on output screen.
- cout is a predefined object
- The symbol (`<<`) is called as insertion operator.
- Both operators are used to display output of program

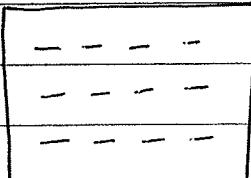
Syntax:

`Cout << "output message";`

or

`Cout << VariableName;`

Screen



26

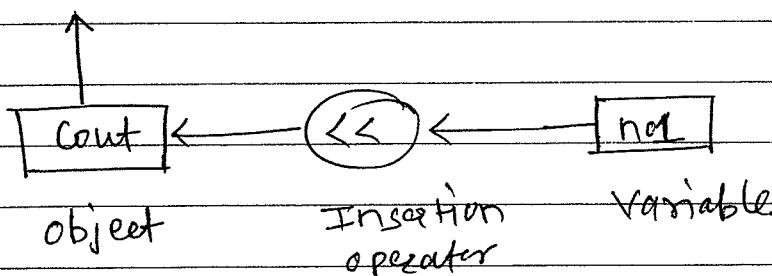


fig: output using cout operator

eg.

- 1) `int a=10;`
- 2) `Cout << "HelloWorld";`
- 3) `float b=10.11;`
- 4) `Cout << b;`

- we can display multiple variable value by using cout object.

syntax:

```
cout << var1 << var2 << var3 ... ;
```

eg:

```
int a=10, b=20, c=30;
```

```
cout << a << "It" << b << "It" << c;
```

2. Objects and classes

* structure :

- structure is a collection of different types of data.
- structure is a group of different data types.
- Each variable's within structure is called as 'member of structure'.
- The name given to structure is called as 'structure tag'.
- struct is a predefined keywords which is used for declaration of structure.

Syntax :

```
struct struct_name  
{  
    // list of diffn data type's  
};
```

e.g.

```
struct student  
{  
    int rollno;  
    float marks;  
    char name[10];  
};
```

Structure Variables :

- memory is allocated for structure when variable is created.

- There are two different ways to create variable of structure.

① Variables are declared immediately after closing curly bracket.

Syntax:

```
struct struct_Name  
{
```

// list of diffn data types.

```
} struct_variable;
```

Eg.

```
struct student  
{
```

int rollno;

float marks;

char name[10];

```
} s1; // s1 is a name of struct Variable
```

② The variables are declared by using struct keyword & structure name.

Syntax:

```
struct struct_Name VariableName;
```

Eg.

```
struct student s1; // s1 is a name  
of struct var.
```

Accessing structure Members:

- structure members are accessed by using dot (.) operator.
- dot (.) operator also called as structure member operator.
- Before the dot structure variable & after the dot member of structure is written.

Syntax:

struct-variable-name . struct-member-name

Eg.:

s1. rollno = 101;

s1. marks = 98.99;

s1. name = "Velskii";

- Each and every structure member allocates individual memory for each variable of structure.

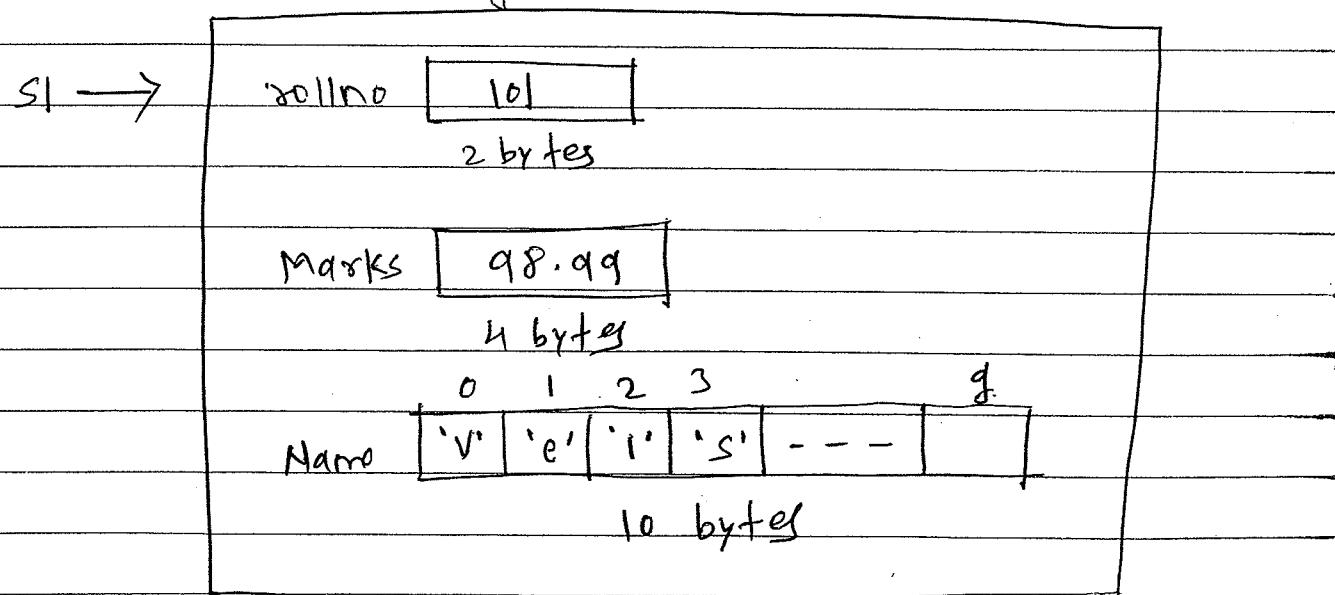


Fig: Memory Representation of structure variable

Program:

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

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

```
struct student
```

```
{
```

```
    int rollno;
```

```
    float marks;
```

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

```
}
```

```
void main()
```

```
{
```

```
    struct student s1;
```

```
    clrscr();
```

```
    cout << "nEnter student rollno:";
```

```
    cin >> s1.rollno;
```

```
    cout << "nEnter student marks:";
```

```
    cin >> s1.marks;
```

```
    cout << "nEnter student Name:";
```

```
    cin >> s1.name;
```

```
    cout << "n *** student information ***";
```

```
    cout << "n Roll No : " << s1.rollno;
```

```
    cout << "n Name : " << s1.name;
```

```
    cout << "n Marks : " << s1.marks;
```

```
    getch();
```

```
}
```

0/p

Enter student rollno : 101

Enter student marks : 98.99

Enter student Name : Velskii

*** Student information ***

RollNo : 101

Name : Velskii

Marks : 98.99

* Difference between structure and class:

structure

i) It is a collection of different types of data

ii) 'struct' keyword is used.

iii) Structure variables can be created

iv) Functions are not present in structure.

v) No private or public member declaration

vi) No data hiding

vii) Syntax: ——————

class

i) class is a collection of data and function.

ii) 'class' keyword is used.

iii) class variables called as objects.

iv) Functions are present in structure.

v) members can be declared as private, public & protected.

vi) Data hiding

vii) Syntax: ——————

* classes:

- class is one of the important feature of C++.
- class is a collection of similar types of objects.
- class contain data & function.
- class is a user defined data type.
- class is a collection of data members & member function.
- class is a predefined keyword.
- class contain private, public or protected members.
- By default visibility of class is private.

Syntax:

```
class class_Name
```

```
{
```

```
private :
```

```
// variable declaration;
```

```
// function declaration;
```

```
public :
```

```
// variable declaration;
```

```
// function declaration;
```

```
}
```

- private & public both are visibility mode of class.

Eg.

```

class student
{
private :
    int rollno;
    float marks;
    char name;

public :
    void getdata ()
    {
        cout << "Enter rollno: ";
        cin >> rollno;
        cout << "Enter Name: ";
        cin >> name;
        cout << "Enter Marks: ";
        cin >> marks;
    }

    void display ()
    {
        cout << "Roll No: " << rollno;
        cout << "Name: " << name;
        cout << "Marks: " << marks;
    }
};

```

- Those members declared as private, it can not be accessible outside the class.
- Those members declared as public, it can be accessible outside the class.
- Data hide: Inside the class due to private keyword.

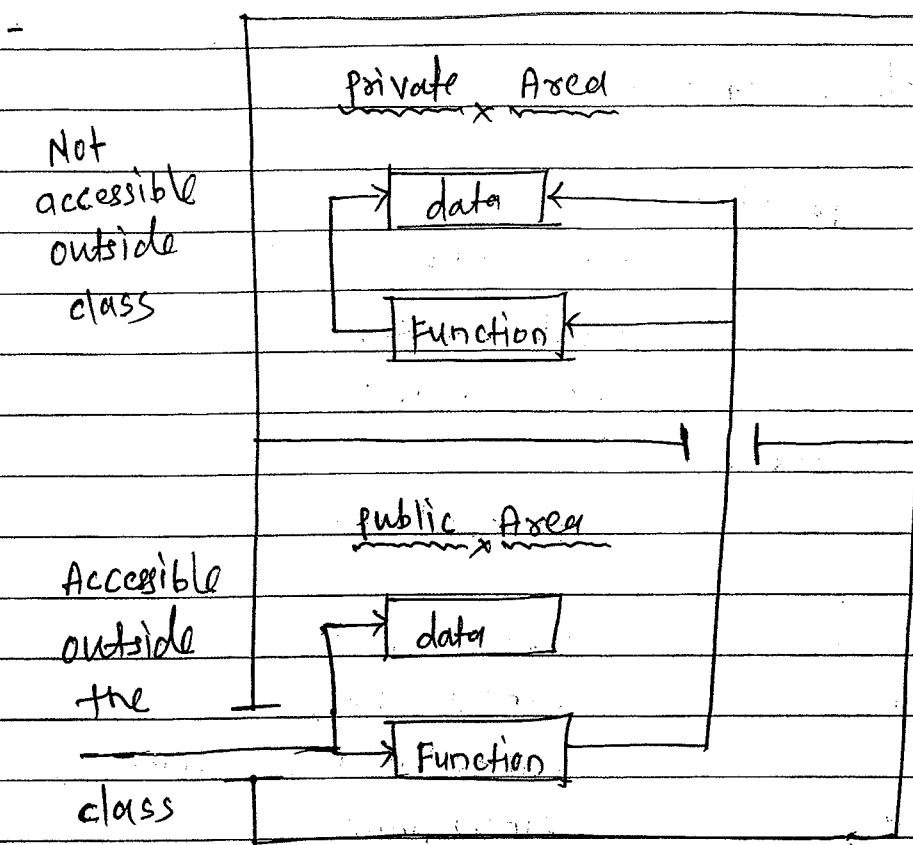


fig: Representation of classes:

* creating objects:

- class variables are known as objects.
- we can create multiple objects from class.
- objects are basic run time entities.
- Syntax:

class-name object-name;

Eg:

student s1; // object created
// from student class.

- when object is created then memory is allocated for each data member's.
- More than one objects can be declared in a single statement.

Eg.

student s1, s2, s3;

- There is another way for declaration of objects.

Eg.

class student
{

} s1, s2, s3; // objects of class.

* Accessing class members:

- once, object of class is created then we can access members of class by using dot(.) operator.

- The private data of a class can be accessed only by us through the member functions of that class.

- Syntax:

objectName. dataMember;

or

objectName. member function (Argument);

Eg.

s1. ~~getdata()~~

s1. ~~display()~~

s1. ~~display()~~

- The objects variables declared as public can be accessed by objects directly.

Eg.

class item

{

private:

int x1;

int y1;

public:

int z1;

}

item a1;

a1.x1 = 10; // invalid, it will give error

a1.z1 = 10; // valid because y1 is public.

* Defining Member function :

- Member functions can be defined in two places.

- ① Inside the class definition.
- ② outside the class definition.

① Inside the class definition :

- In this case function definition written inside the class.

- When function definition written inside the class then it is treated as inline function.

Eg:

```
class Addition
{
```

private:

```
int a,b,c;
```

public:

```
void getdata() // inside fun defn
{
```

```
a=10;
```

```
b=5;
```

```
void display() // inside fun defn.
{
```

```
c=a+b;
```

```
cout << "Addition = " << c;
```

```
}
```

```
}
```

② outside the class definition:

- Member function declared inside the class and its definition written outside the class.
- while writing function definition outside the class then we need to use classname and scope resolution operator (`::`).

Syntax:

```
returntype classname :: function-name(arg. list)
{
    // Function body;
}
```

Eg.

```
class Addition
{
private:
    int a, b, c;
public:
    void getdata(); // funn declaration
    void display(); // funn declaration
}
```

```
void Addition :: getdata() // funn definition outside
{
    a = 10;
    b = 5;
}
```

```
Void Addition:: display()
```

{

```
    c = a + b;
```

```
    cout << "in Addition = " << c;
```

}

```
Void main()
```

{

```
    Addition a1;
```

```
    a1.getdata();
```

```
    a1.display();
```

```
    getch();
```

}

O/P	
	Addition = 15

- class name and scope resolution operator is used to tell the compiler the scope of the member functions.

* Array of objects:

- Array is a collection of similar type of data.
- Array of variables of created from class is known as "Array of objects".
- Instead of creating multiple objects, we can create array of objects.

Syntax:

```
class Name object_Name [size];
```

Eg:

```
class Addition
```

```
{
```

```
private:
```

```
int a, b;
```

```
public:
```

```
void getdata()
```

```
{
```

```
cout << "Enter two numbers: "
```

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

```
}
```

```
void display()
```

```
{
```

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

```
}
```

```
void main()
{
```

```
    Addition a1[3]; // Array of objects.  
    clrscr();
```

```
    for (int i=0; i<3; i++)  
    {  
        a1[i].getdate();  
    }
```

```
    for (int i=0; i<3; i++)  
    {  
        a1[i].display();  
    }  
    getch();  
}
```

O/P

Enter two numbers: 10 20

Enter two numbers: 50 100

Enter two numbers: 100 200

Addition = 30

Addition = 150

Addition = 300

a	10	{ a1[0]
b	20	}
a	50	{ a1[1]
b	100	}
a	100	{ a1[2]
b	200	}

fig: storage of data items of an object array

* Memory Allocation for objects:

- When class is declared no memory is allocated for data members.
- Memory is allocated for objects when they are declared.
- Data members allocate separate or individual memory space for each objects.
- Member functions are created & allocated space in memory only once when they are defined in the class.

Eg.:

class Sample

{

private:

int x, y;

public:

void getdata()

{ x = 20;

y = 30;

}

Sample s_1, s_2 ;

- In above example s_1 & s_2 both are objects.
- Each object has two data members x & y .
- So total 4 bytes memory allocated for each objects.

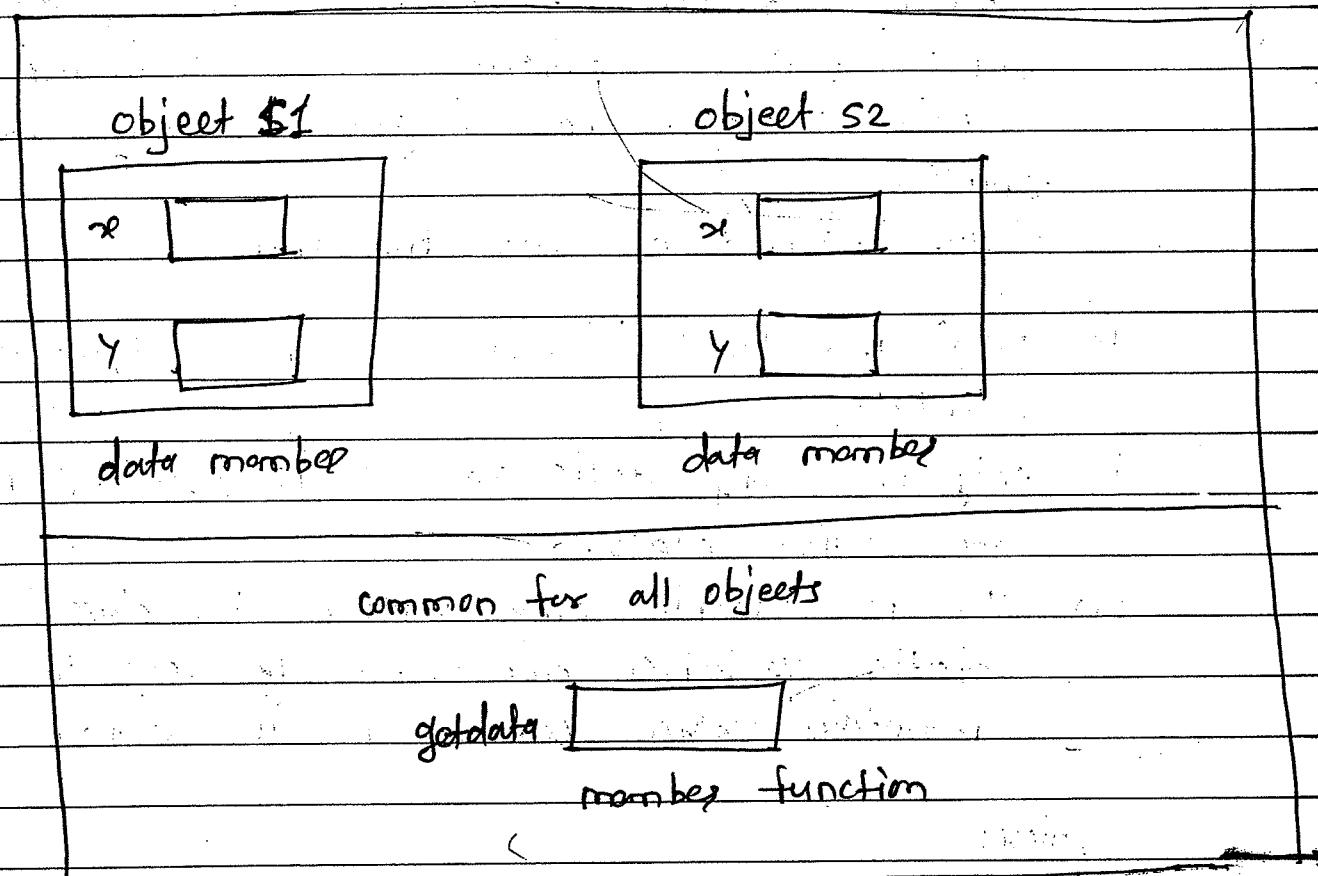


fig: objects in memory

- All member functions are stored in memory only once and all objects share member function.

* Static Data Members:

- we can create data members as static by using static keyword.
- static is a predefined keyword.
- static memb variables initialized to zero when object is created.
- static variable stored in memory only once and all objects share commonly.
- static variable not belongs to any particular object.
- characteristics of data members:
 - i) It is initialized to zero when first object is created.
 - ii) only one copy¹ static variable is created for the entire class.
 - iii) It is visible only within the class.
 - iv) static variables are normally used to maintain values common to the entire class.

Syntax:

```
class className
{
    static datatype variablename; // static variable
    public: // declaration
    -----
}

datatype className :: staticvariablename; // static
// variable definition.
```

- static variable definition needs to be defined outside the class.

program:

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

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

```
class item
```

```
{
```

```
private :
```

```
static int count; // Count is static  
int no;
```

```
public :
```

```
void getdata (int x)
```

```
{
```

```
no = x;
```

```
count++;
```

```
}
```

```
void display ()
```

```
{
```

```
cout << "In count = " << count;
```

```
}
```

```
};
```

```
int item :: count; // static variable count is defined.
```

```
Void main()
```

```
{
```

```
item x1, y1, z1;
```

`c1.display();`

`x1.display();`

`y1.display();`

`z1.display();`

`x1.getData(10);`

`y1.getData(20);`

`z1.getData(30);`

`x1.display();`

`y1.display();`

`z1.display();`

`getdata();`

}

O/P

Count = 0

Count = 0

Count = 0

Count = 3

Count = 3

Count = 3

object x1

No [10]

object y1

No [20]

object z1

No [30]

count [3]

common to all three
objects.

* Static Member Function :

- member functions may also be declared as static.
- Syntax:

```
static returnType funname (arg. list)
{
    // body of fun^n
}
```

- Characteristics of static member function:

- A static member function can access only other static members.
- static member function can be called using class name.
- They may not be virtual.
- They do not have 'this' pointer.
- There cannot be static and non-static version of the same function.

Program:

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



```
class item
```

```
{
```

```
private:
```

```
    static int count;
```

```
    int no;
```

```
public:
```

```
    void getdata (int x)
```

```
{
```

```
    no = x;
```

```
    count++;
```

```
}
```

```
    static void display () // static member fun
```

```
{
```

```
    cout < " " >> count = " << count;
```

```
}
```

```
};
```

```
int item :: count;
```

```
void main()
```

```
{
```

```
    item x1, y1, z1;
```

```
    clrscr();
```

```
    x1 ::
```

```
    item :: display (); // call static function
```

```
    x1 :: getdata (10);
```

```
    y1 :: getdata (20);
```

```
    z1 :: getdata (30);
```

```
    item :: display (); // calling static function
```

```
    getch();
```

```
}
```

O/P

count = 0

count = 3

* Friend Functions:

- friend is a predefined keyword.
- The private members can not be accessed from outside the class.
- Sometimes there could be situation where we want to share one function betn two classes.
- In that case we can make that function as friend function.
- friend function access private data of class.
- friend function not in scope of class.
- syntax:

class className
{

public:

friend returnType funName(arguments);

}

- friend keyword written before the fun decl.

- No need to use class name & scope resolution operator while writing friend fun defn outside the class.

- characteristics of Friend function:

- i) It is not in the scope of the class.
- ii) It cannot be called by using object. It can be invoked like normal function.
- iii) It can be declared either in public or private.
- iv) It has the objects as arguments.
- v) object name and dot operator used for accessing data members inside the friend function.

program :

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

class Sample

{

int a, b;

public:

void getdata()

{

a = 20;

b = 30;

}

```
friend float mean( sample s1 ); // friend fun  
declaration  
};
```

```
float mean( sample s1 )  
{
```

```
return((s1.a + s1.b)/2);  
}
```

```
void main()  
{
```

```
sample x;  
clrscr();  
x.getdata();  
cout << "mean value = " << mean(x);  
getch();  
}
```

O/P

Mean value = 25.0

3. Constructors & Destructors.

PAGE NO. _____
DATE 18/03/2017

* Constructor:

- constructor is a special member function of class.
- constructor is used to initialize objects of its class.
- constructor name & class name both are same.
- There is no return type for constructor even void also.
- constructor is invoked when objects of class is created.
- There is no need to call constructor by using object name & dot(.) operator.
- There are three types of constructors
 - 1) Default constructor
 - 2) parameterized constructor.
 - 3) copy constructor.

Syntax:

Characteristic of constructor:

- i) constructor name & class name both are same.
- ii) They do not have return types
- iii) constructor(s) should be present in public section.
- iv) It called automatically when object is created.
- v) Constructor can not be inherited.
- vi) Constructor can not be virtual.
- vii) It is used to initialize object of class.
- viii) constructor is a special member function of class.

* Default constructor:

- when constructor does not takes any arguments then it is called as default constructor.
- A constructor that accept no arguments is called default constructor.
- Constructor is used to create object of class
- If we don't write constructor in program then compiler supplies default constructor.
- If we provide/write constructor in program then compiler will not create default constructor.
- default constructor name is of class name both are same.
- constructor is called when first object of class is created.
- There is no return type for constructor.

Syntax:

class class-name

{

 -- -

public:

 class-name(); // default constructor

 { }

 // constructor - It can do anything

 // function - It can do anything

 // body - It can do anything

};

Eg.

class Addition

{

int a, b; c;

public:

Addition() // default constructor

{

a = 10;

b = 5;

}

}

- In above example `Addition()` is default constructor
program:

#include <iostream.h>

#include <conio.h>

class Addition

{

int a, b, c;

public:

Addition()

{

a = 10;

b = 5;

}



Void display()

{

c = a + b;

cout << "in Addition = " << c;

}

}

Void main()

{

Addition a1;

clrscr();

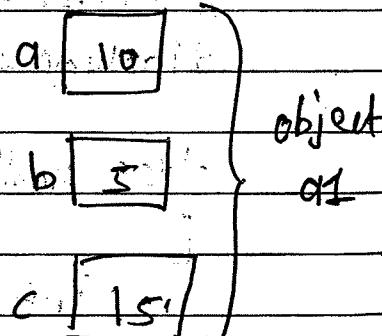
a1.display();

getch();

}

0/0

Addition = 15



* Parameterized constructor:

- parameterized constructor is one of the types of constructor.
- when constructor takes some argument then it is called parameterized constructor.
- A constructor which accept arguments is called as parameterized constructor.
- constructor name & class name both are same.
- there is no any return type for constructor.

Syntax:

class className

{

public:

className (parameter lists)

{

}

}

Eg.:

class Addition

{

int a, b, c;

public:

Addition (int x, int y) // parameterized constructor

{ a = x;

$b = y;$

}

}

- In above example Addition (int x, int y) is parameterized constructor.

Program:

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

class Addition
{
    int a, b, c;
public:
    Addition (int x, int y)
    {
        a = x;
        b = y;
    }
    void display()
    {
        c = a + b;
        cout << "In Addition = " << c;
    }
}
```

}

Void main()

{

Addition a1(10,5); // Pass arguments to constructor.

class A

a1.display();

getdata();

}

Output

Addition = 15

- we can pass arguments to parameterized constructor by using two way's.

- ① By calling constructor implicitly:

Addition a1(10,5); // implicit call

- ② By calling constructor explicitly:

Addition a1 = Addition(10,5); // Explicit call

* COPY CONSTRUCTOR :

- A constructor can accept a reference to its own class as a parameter is called as a copy constructor.
- When constructor takes object as parameter is known as copy constructor.
- A copy constructor can be used to declare & initialize an object of another object.

Eg:-

Time T2(T1);

- In above example, define the object T2 & at the same time initialize it by using values of T1 object.

Syntax:

class className

{ public: (declaration) }

public: (declaration)

{

}

program :-

#include <iostream.h>

#include <conio.h>

class Time

{

int hrs, min, sec;

public:

Time (int h, int m, int s)

class Code

{

int id;

public:

Code () // default constructor

{

id = 10;

}

Code (Code & x) // copy constructor

{

id = x.id;

}

void display()

{

cout << id;

}

};

Void main()

{

code c1; // default constructor called

code c2(c1); // copy constructor called

close();

code

cout << "object c1 ID:";
c1.display();

cout << "object c2 ID:";
c2.display();

getch();

}

O/P

object c1 ID: 10

object c2 ID: 20

ANSWER

* Constructor overloading:

- when constructor name are same but it's arguments are different then it is called as constructor overloading.
- we can use no argument constructors, one arguments constructors & so on.
- C++ permits to use all constructors in the same class. then it becomes constructor overloading.

Eg:

```

class Code
{
    int id;
public:
    Code()           // default constructor
    {
        id=100;
    }
    Code(int x)     // parameterized constructor
    {
        id=x;
    }
    Code(Code fm)   // copy constructor
    {
        id=fm.id;
    }
}

```

void display()

{

cout << id;

}

}

Void main()

{

Code c1;

Code c2(200);

Code c3(c2);

cout << "In object c1 id :";

c1.display();

cout << "In object c2 id :";

c2.display();

cout << "In object c3 id :";

c3.display();

getchar();

}

A/P

object c1 id: 100

object c2 id: 200

object c3 id: 200

- In above example, we can see first constructor takes no argument, second constructor takes one argument & third constructor takes object as argument.
- Constructor name 'code' is same but it's arguments are different then it is called as constructor overriding.

* Constructors with Default Arguments:

- It is possible to define constructor with default argument.

- Eg:

constructor Addition() can be declared as follow.

Addition (int x, int y=0);

- the default value of the argument y is zero.

- if we write the below statement:

Addition a1(10,5);

- Assign 10 to x and 5 to y variable.

- If we specify actual parameter value then default value is override.

program:

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

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

```
class Addition
```

```
{
```

```
    int a,b,c;
```

```
public:
```

Addition (int x, int y=0) // constructor with default value.

{

a = x;

b = y;

}

Void display()

{

c = a + b;

cout << "Addition = " << c;

}

getch();

};

void main()

{

Addition a1;

a1.display();

a1.display();

getch();

}

O/P

Addition = 15

* Destructor:

- Destructor is used to destroy the objects which is created by using constructor.
- Destructor also special member function of class.
- Destructor name of class name both are same but destructor is preceded by tilde (~) operator.
- Destructor does not takes any arguments.
- It does not returns any value.
- Destructor will be invoked implicitly by the compiler when program comes out from closing curly bracket.
- Destructor is used to destroy the objects & free the memory.
- We can not invoke / call constructor explicitly.
- Destructor also present in public section of class.
- Destructor can not be inherited.
- Syntax:

className

{

public:

\sim className() // destructor

{

}

}

Eg:

int count = 0;

class Alpha

{

public:

Alpha() // constructor

{

Count++;

cout << "No of object created: " << Count;

}

\sim Alpha() // Destructor

{

cout << "No of object destroyed: " << Count;

Count--;

}

}

```
Void main()
```

```
{
```

```
    classx();
```

```
    Alpha a1, a2;
```

```
{
```

```
    Alpha a3;
```

```
}
```

```
{ Alpha a4;
```

```
    getx();
```

```
}
```

O/P

No of object created : 1

No of object created : 2

No of object created : 3

No of object destroyed : 3

No of object created : 3

No of object destroyed : 3

No of object destroyed : 2

No of object destroyed : 1

X

Difference b/w constructor and Destructor

constructor

1) Constructor is used to create objects.

2) Its name is same as class name.

3) Constructor can take arguments.

4) Constructor is involved when object is created.

5) Eg.

Addition()

{

 --

}

--

--

Destructor

1) Destructor is used to destroy the objects.

2) Its name is same as class name but preceded by tilde (~).

3) Destructor does not take any arguments.

4) It is involved by compiler when program reaches closing curly bracket.

Eg:

~Addition()

{

}

4. Inheritance

PAGE NO. 18 DATE 03/20/17

* Inheritance:

- Inheritance is one of the major important properties of OOP.
- To create new class by using the properties of old class is known as inheritance.
- Newly created class is known as Derived class.
- Old class is known as Base or super class.
- Acquire the properties of base class into derived class is known as inheritance.
- Inheritance will shows the reusability because of inheritance development time will save & reduce project cost.
- The mechanism of deriving new class from old class is known as inheritance.

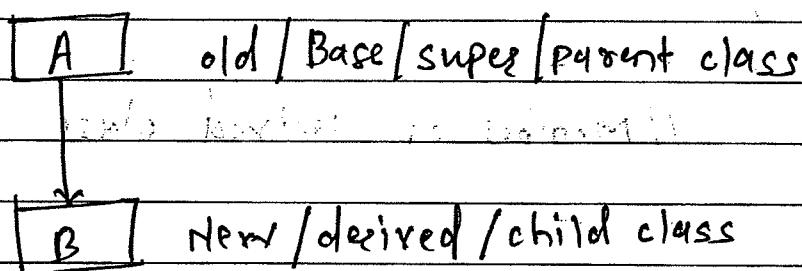


fig: inheritance

- Class B is created from old class A is known as inheritance.

Types of inheritance:

- 1) single inheritance
- 2) multilevel inheritance
- 3) multiple inheritance
- 4) Hierarchical inheritance
- 5) Hybrid inheritance.

* Derived classes:

- The mechanism of deriving new class from old is called inheritance.
- Derived class is defined by specifying relationship with base class.
- Syntax:

```
class derived-class : visibility-mode base-class  
{
```

II Members of derived class

3

- The colon indicates that derived class name is derived from base class
- visibility mode is optional
- If present, then it may be either private or public.

- Visibility mode specifies how the features of base class is inherited into derived class.

- Eg.

class ABC: private xyz // private derivation

{
 // members of ABC
}

}

class ABC: public xyz // public derivation

{
 // members of ABC
}

}

class ABC: xyz // private derivations by default.

{
 // members of ABC
}

}

- Visibility modes & their effects:

Base class visibility	Derived class visibility	
	public	private
private	Not inherited	Not inherited
public	public	private
protected	protected	private.

* Member Declaration: protected:

- The private member of a base class cannot be inherited in derived class.
- The public member of base class can be inherited in derived class.
- Public members are accessible everywhere in the program
- C++ provides third visibility modifiers - protected
- A member declared as protected is accessible in same class & it's immediate subclass.
- It cannot be accessed by the functions outside these two classes.

Eg.

class Alpha

{

private :

- // visible only within same class where it is declared.

public :

- // visible everywhere in program

protected :

- // visible in same class & it's derived class.

3/

- when protected member is inherited in public mode then it becomes protected in derived class.
- when protected member is inherited in private mode then it becomes private in derived class.

* single inheritance:

- A derived class from only one base class is called as single inheritance.
- single inheritance is one of the types of inheritance.

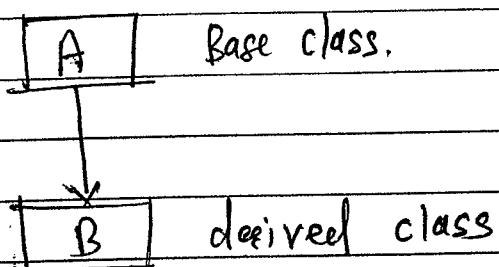


fig: single inheritance

- In above figure, we can see that derived class B is created from only one base class A.

Syntax:

class derived_class : visibility-mode Base-class

{

//members of derived class

}

Eg:

#include <iostream.h>

#include <conio.h>

class A

{

protected:

int a, b;

public:

void getdata()

{

cout << "Enter two values:";

cin >> a >> b;

}

}

class B : public A

{

protected:

int c;

public:

void display()

{

c = a + b;

cout << "Addition = " << c;

}

}

Void main()

{

B obj;

clrscr();

obj.getdata();

obj.display(); getch();

}

Q1e

Enter two values:

10

5

Addition = 15

class A

protected: a, b

public: getdata()

↓
public derivation.

class B

protected:

a, b, c

public:

getdata()

display()

* Multilevel Inheritance:

- The mechanism of deriving a class from another derived class is known as multilevel inheritance.
- To create new class from another derived class is known as multilevel inheritance.

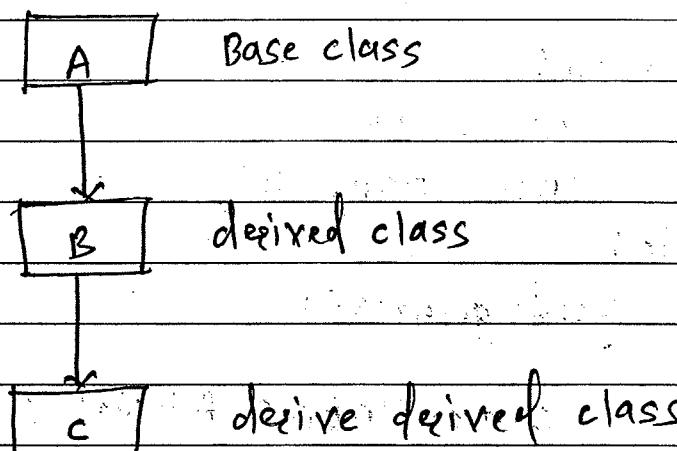


fig: Multilevel Inheritance

- Above figure shows class C is derived from another derived class B.
- Multilevel inheritance is implemented as below.

class A // Base class A

{ ---
--- }

}

class B : public A // B derived from A

{ ---
--- }

}

class C : public B // C derived from B

{ ---
--- }

}

program:

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

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

```
class student
```

```
{
```

```
protected:
```

```
int rollno;
```

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

```
public:
```

```
void getdata()
```

```
{
```

```
cout << "Enter roll No: ";
```

```
cin >> rollno;
```

```
cout << "Enter Name: ";
```

```
cin >> name;
```

```
}
```

```
void putdata()
```

```
{
```

```
cout << "roll No: " << rollno;
```

```
cout << "Name: " << name;
```

```
}
```

```
}
```

```
class Test : public student
```

```
{
```

```
protected:
```

```
int mark1, mark2;
```

```
public:
```

void getdata1()

{

cout << "Enter mark1 :";

cin >> mark1;

cout << "Enter mark2 :";

cin >> mark2;

}

void putdata1()

{

cout << "\n mark1 : " << mark1;

cout << "\n mark2 : " << mark2;

}

}

class Result : public Test

{

protected:

int total;

public:

void display()

{

total = mark1 + mark2;

cout << "\n Total = " << total;

}

}

void main()

{

Result R1;

R1.display();

P1 : getdata();

P1 : getdata1();

P1 : putdata();

P1 : putdata1();

P1 : display();

getch();

}

O/P

Enter RollNo : 101

Enter Name : James

Enter mark1 : 98

Enter mark2 : 36

RollNo : 101

Name : James

Mark1 : 98

Mark2 : 36

Total = 134

* Multiple inheritance:

- The mechanism of deriving a class from more than one base class is known as multiple inheritance.
- To create new class from more than one old class is known as multiple inheritance.

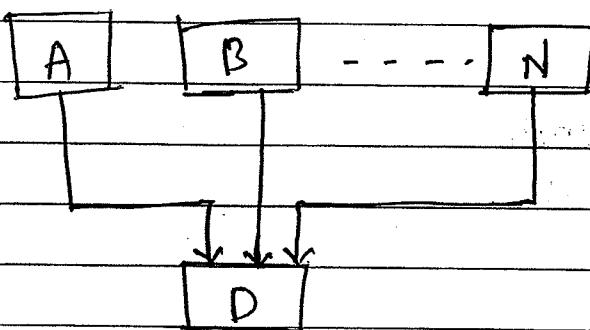


fig: multiple inheritance.

- Above fig. shows class D is derived from base class A, B, ..., etc.

Syntax:

```
class derived-class : visibility-mode Base1, visibility-mode Base2
{ }
```

// Members of derived class

3)

- Base classes are separated by commas.
- In multiple inheritance one problem will come, if same method name present in two different base classes but new class

is created from those classes then it will create ambiguity.

- Compiler will confuse which function to be called.
- we can solve this problem by using Virtual base class concept.

Program:

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

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

```
class M
```

```
{
```

Protected:

```
int a;
```

public:

```
void get-a()
```

```
{
```

```
cout << "Enter value of a:";
```

```
cin >> a;
```

```
}
```

```
void put-a()
```

```
{
```

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

```
}
```

Program: This program shows how to implement virtual base class.

This program has two derived classes.

Both classes inherit from a common base class.

The base class has a protected member variable 'a'.

class N

{

protected:

int b;

public:

void get-b()

{

cout << "nEnter value of b:";

cin >> b;

}

void put-b()

{

cout << "n b = " << b;

}

};

class P : public M, Public N

{

protected:

int c;

public:

void display()

{

c = a + b;

cout << "n Addition = " << c;

}

};

Void main()

{

P p1;

c1*scr();

p1.get-a();

p1.get-b();

ex:displ

p1.put-a();

p1.put-b();

p1.display();

getch();

}

O/P

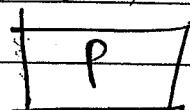
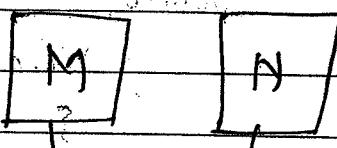
Enter value of a: 10

Enter value of b: 5

a=10

b=5

Addition=15



* Hierarchical Inheritance:

- To create multiple derived classes from only one base class is known as Hierarchical inheritance.

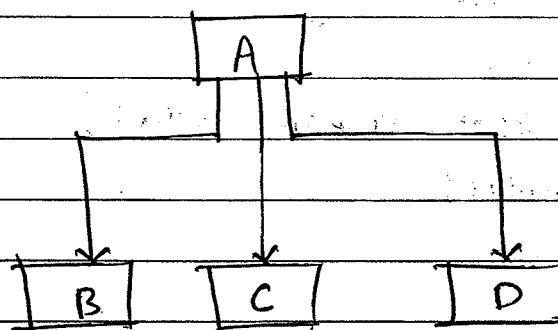


fig: Hierarchical Inheritance

- Above fig. shows derived class B, C & D is created from base class A.

- If we want to share features of one level to another sub level then we can use Hierarchical inheritance.

- In this inheritance, we can create multiple subclasses from only one base class.

Program:

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

```
class Employee
```

```
{
```

```
protected :
```

```

int emp_id;
char emp_name[10];
public:
void getdata()
{
    cout << "Enter employee id: ";
    cin >> emp_id;
    cout << "Enter employee Name: ";
    cin >> emp_name;
}

```

```

void putdata()
{
    cout << "EMP id: " << emp_id;
    cout << "EMP name: " << emp_name;
}

```

```
};
```

```
class worker : public Employee
```

```
{
protected:
```

```
int overtime_sal;
```

```
public:
```

```
void getdata()
```

```
{
    cout << "Enter overtime salary: ";
    cin >> overtime_sal;
}
```

```
void putdata()
```

```
}
```

cout << "Enter overtime salary: " << overtime_sal;

}

}

class Manager : public Employee

{

protected:

int Additional_allowance;

public:

void getdata2()

{

cout << "Enter Additional Allowance: ";

cin >> Additional_allowance;

}

void putdata2()

{

cout << "Additional Allowance: " <<
Additional_allowance;

}

}

void main()

{

Worker w1;

Manager m1;

clerk c1;

cout << "Enter Worker details: ";

w1.getdata();
w1.getdata1();

cout << "Enter workers manager details:";

m1.getdata();
m1.getdata2();

cout << "Enter workers information :";

w1.putdata();
w1.putdata1();

cout << "Enter Manager Information :";

m1.putdata();
m1.putdata2();

getch();

3

O/P

Enter workers details:

Enter employee id: 148464

Enter employee Name: Rishabh

Enter overtime salary: 800

Enter Manager details:

Enter employee id: 32956

Enter employee Name: Nilesh

Additional allowance: 1500

*** Worker Information ***

EMP id : 118464

EMP Name : Vishal

Overtime salary : 800

*** Manager Information ***

EMP id : 32956

EMP Name : Nitish

Additional Allowance : 1500

* Hybrid Inheritance :

- The combination of more than one types of inheritances is known as Hybrid Inheritance.
- When more than one type of inheritance is used in any design, it is called as hybrid inheritance.
- There could be situation where we need to use two or more than types of inheritance to solve problem.

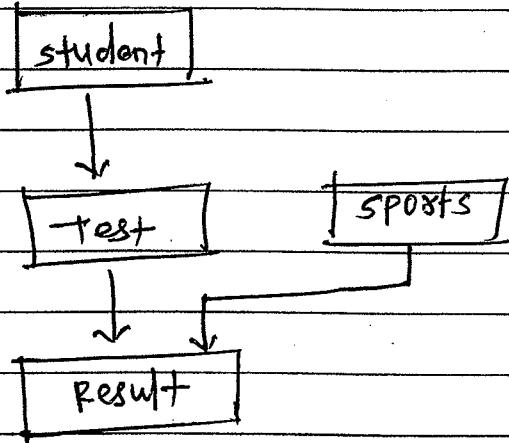


fig: Hybrid Inheritance

- Above fig. shows the combination of two types of inheritance, one is multilevel & second is multiple.

Program:

#include <iostream>

program:

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

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

which now

```
class student
```

```
{
```

protected:

```
int rollno;
```

public:

```
void getdata()
```

```
{
```

```
cout<<"\n Enter Roll No:";
```

```
cin>>rollno;
```

```
}
```

```
void putdata()
```

```
{
```

```
cout<<"\n Roll No: " << rollno;
```

```
}
```

```
};
```

```
class Test : public student
```

```
{
```

protected:

```
int mark1, mark2;
```

public:

```
void getdata1()
```

```
{
```

```
cout<<"\n Enter Mark1:";
```

```
cin>>mark1;
```

```
cout << "Enter Mark2:";
```

```
cin >> mark2;
```

{

```
void putdata2()
```

{

```
cout << "Mark1 = " << mark1;
```

```
cout << "Mark2 = " << mark2;
```

{

{

```
class sports
```

{

```
protected:
```

```
float sportwt;
```

```
public:
```

```
void getdata()
```

{

```
cout << "Enter sport weight:";
```

```
cin >> sportwt;
```

{

```
void putdata()
```

{

```
cout << "sport weight = " << sportwt;
```

{

{

```
class Result { public Test, public sports.
```

{

```
protected:
```

```
    int total;
```

```
public:
```

```
    void display()
```

{

```
        total = mark1 + mark2;
```

```
        cout << "In total = " << total;
```

}

};

```
void main()
```

{

```
    Result R1;
```

```
    R1.display();
```

```
R1.getdata();
```

```
R1.getdata1();
```

```
R1.getdata2();
```

```
R1.putdata();
```

```
R1.putdata1();
```

```
R1.putdata2();
```

```
R1.display();
```

```
getdata();
```

}

e/p

Enter roll No : 101

Enter mark1 : 98

Enter mark2 : 90

Enter sport weight : 23.98

roll No : 101

mark1 : 98

mark2 : 90

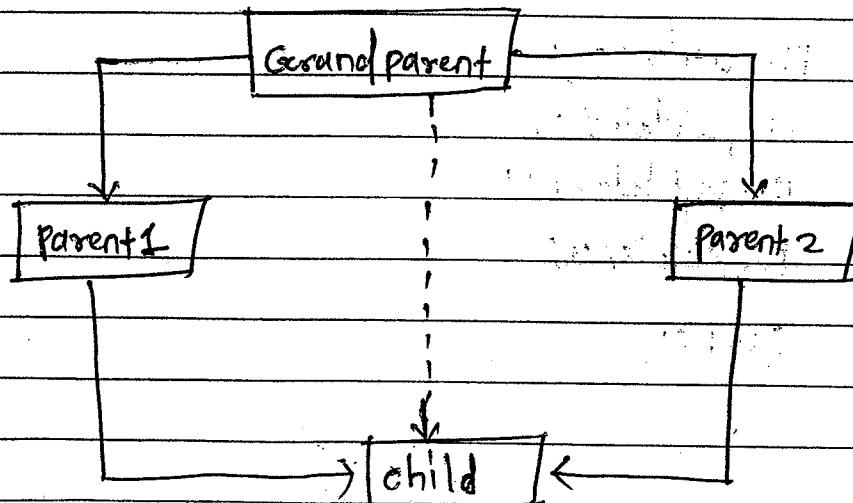
sport weight : 23.98

Total = 188



* Virtual Base class :

- x x x
- Virtual is a predefined keyword.
 - sometimes in situations, all three types of inheritance multilevel, multiple & hierarchical are involved.
 - This is shown in below figure:



- In above figure child has two base classes parents of parentz which is derived from common base class Grandparent.
- Child class inherits the properties of Grandparent class from parents of parentz.
- It means members of grandparent class present twice in child class.
- Compiler gets confuse which method will call.
- To avoid this problem, we can use virtual base class concept.

Eg.

```
class A // grandparent
{
```

}

```
class B1 : virtual public A // parent1
```

{

}

```
class B2 : public virtual A // parent2
```

{

}

```
class C : public B1, public B2 // child
```

{

// only one copy of A will be

// inherited.

}

- when class is made a virtual base class, then C++ takes care about only one copy of that class is inherited.
- The keywords `virtual` & `public` may be used in any order.

Program:

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

class student
{
protected:
    int rollno;
public:
    void getdata()
    {
        cout<<"Enter Roll No.:";
        cin>>rollno;
    }

    void putdata()
    {
        cout<<"Roll No.:"<<rollno;
    }
};
```

class test : public virtual student

{

protected:

int mark1, mark2;

public:

void getdata1()

{

cout << "Enter Mark1:";

cin >> mark1;

cout << "Enter Mark2:";

cin >> mark2;

}

void putdata1()

{

cout << "Mark1 = " << mark1;

cout << "Mark2 = " << mark2;

}

};

class sports : virtual public student

{

protected:

int sportwt;

public:

void getdata2()

{

cout << "Enter sport weight:";

cin >> sportwt;

}

```
void putdata()
```

{

```
cout << "In sport weight = " << sportwt;
```

}

};

```
class Result : public test, public sports
```

{

```
protected:
```

```
int total;
```

```
public:
```

```
void display()
```

{

```
total = mark1 + mark2;
```

```
cout << " In Total = " << total;
```

}

};

```
void main()
```

{

```
Result R1;
```

```
R1.Sports();
```

```
R1.getdata();
```

```
R1.getdata1();
```

```
R1.getdata2();
```

```
R1.putdata();
```

```
R1.putdata1();
```

R1.putdata2();

R1.display();

getdata();

}

O/P

Enter roll No: 101

Enter mark1: 98

Enter mark2: 70

Enter sport weight: 70.17

roll No: 101

mark1 = 98

mark2 = 70

sport weight = 70.17

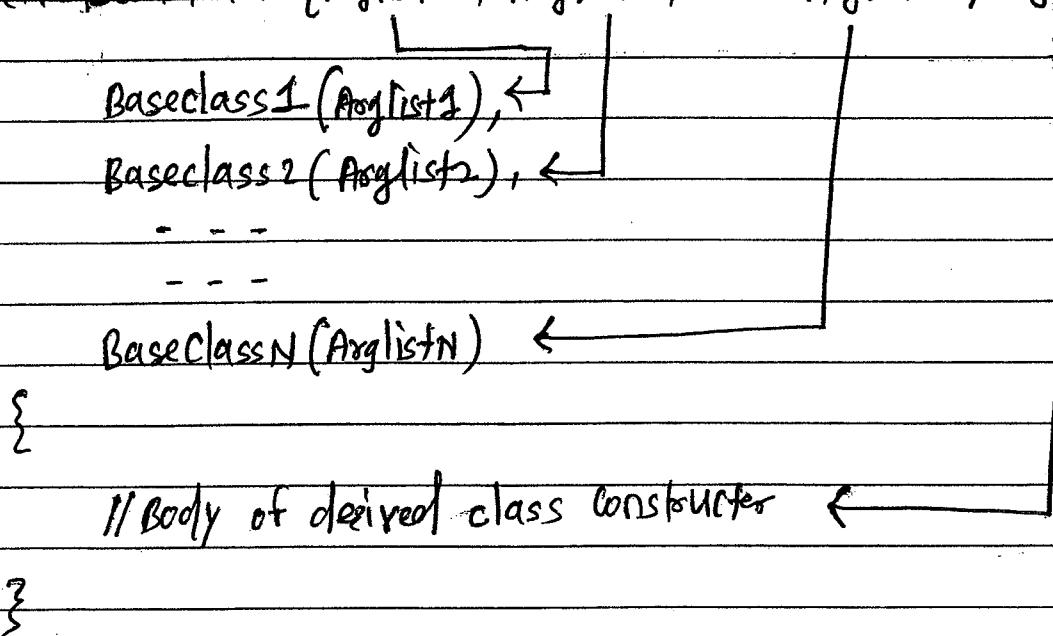
Total = 168

* Constructors in Derived class:

- Constructors are used to initialize objects.
- If base class constructor does not contain then there is no need to write derived class constructor.
- But if base class constructor contain constructor then we need to write derived class constructor.
- Derived class constructor is used to invoke base class constructor.
- Base class constructor executed first before the derived class constructor.
- Derived class constructor supply arguments to the base class constructor.

Syntax:

derived constructor(Arglist1, Arglist2, --- ArglistN, ArglistD):



- Derived class constructor receives the entire list of arguments & passes them to the base constructor.
- The base class constructor executed before executing the statement of derived class :- constructor.

Program:

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

```
class Alpha
```

```
{
```

```
    int x;
```

```
public:
```

```
    Alpha(int i)
```

```
{
```

```
    x = i;
```

```
}
```

};

```
void show_x()
```

```
{
```

```
    cout << "In x = " << x;
```

```
}
```

};

```
};
```

```
class Beta
```

```
{
```

```
    int y;
```

```
public:
```

Beta (int i)

{

y = i;

}

void show-y()

{

cout << "In y = " << y;

}

};

class Gamma : public Beta, public Alpha

{

int m, n;

public:

Gamma (int a, int b, int c, int d) : Alpha(a),
Beta(b)

{

m = c;

n = d;

}

void show-mn()

{

cout << "In m = " << m;

cout << "In n = " << n;

}

};

Void main()

{

 Gramma g(10, 20, 30, 40);
 cls();

 g.show-x();

 g.show-y();

 g.show-mn();

 getch();

}

o/p

x = 10

y = 20

m = 30

n = 40

5. Pointers in C++

* Pointers:

- Pointer is a variable which stores memory address of another variable.
- If pointer variable contain memory address of another variable, it means first variable points to second variable.

Syntax:

datatype * ptr-Var-Name;

where: datatype is a type of pointer variable.

- * is a pointer operator.

Eg.:

int * p;

- In above example p variable is an integer type of pointer variable.
- We can declare integer, characters & float type of pointer variable.
- A pointer variable name must be preceded by an asterisk (*).
- We can initialize pointer operator by using below syntax:

ptr-Var-Name = & Variable-Name

Eg.

```
int a=10;  
int *p;
```

a 10

2000 ← memory
address

p = &a;

p 2000

3000

- In above example, we can see pointer variable p holds memory address of a variable.
- the symbol (&) is known as Address Operator.

Program:

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

```
Void main()  
{
```

```
int a=10, *p;
```

```
clrscr();
```

```
p = &a;
```

```
cout << p << endl; cout << "a=" << a; cout << endl;  
cout << p << endl; cout << *p << endl; cout << (*p); cout << "b=" << b;
```

```
getch();
```

```
}
```

O/P

a = 10

*p = 10

Advantages of pointer:

- 1) It allows to pass variables, arrays, functions, string of structures as function arguments.
- 2) It supports dynamic memory allocation & deallocation of memory.
- 3) Pointers improve efficiency of programs.
- 4) Using pointers, variables can be swapped without physically moving them.
- 5) It allows to establish link between data elements of object.

* Pointer Arithmetic:

- pointer is a variable which holds the memory address of another variable.
- some arithmetic operations can be performed with pointers.
- C++ supports four arithmetic operators:
 - i) Addition (+)
 - ii) Subtraction (-)
 - iii) Incrementation (++)
 - iv) Decrementation (--)

Eg:

```
int value, *ptr;
```

```
value = 120;
```

```
ptr = &value;
```

cout << "Before incrementation" << ptr;

cout << ("Address of

cout << (*ptr);

ptr++;

cout << "After incrementation" << ptr;

cout << (*ptr); cout << ptr;

Value	120	ptr	2000
	2000		3000

- Points Variable `ptr` originally holds 2000 memory address of variable `value`.
- But after incrementation, it contains 2002 address.
- Because `++` operator increase the address by size of data types.

Program:

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

Void main()

{

```
int value, *ptr;
clrscr();
value = 120;
```

`ptr = &value;`

`cout << "Memory address before incrementation = ";`
`cout << ptr;`

`ptr ++;`

`cout << "In Memory address after incrementation = ";`
`cout << ptr;`

`getch();`

}

O/P

Memory address before Incrementation: 2000

Memory address after incrementation: 2002

Value	[120]	Pt	[2002]
	2000		3000

- initially pt variable contain 2000 memory address.

- but after incrementation it becomes 2002.

Pointers are not permitted to perform the following arithmetic operations:

- i) multiply or divide
- ii) Bitwise shift.
- iii) Add or subtract type float or double.

* Pointers to Arrays:

- we can use pointer concept with array.
- consider the declaration,

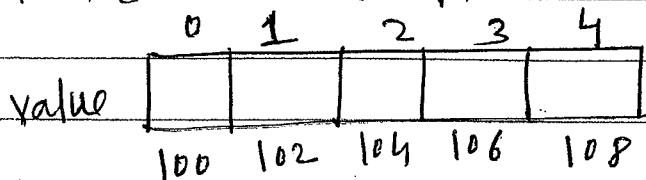
```
int value[20];
```

```
int *ptr;
```

- Assign base address of array to pointer variable.

```
ptr = &value[0];
```

- The address of 0th element is assigned to a pointer variable ptr.



```
ptr [100]
```

- If the pointer is incremented then it will point to next element.

- If we increment or decrement pointer variable then it is incremented or decremented by it's data type size.

Program:

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

```
void main()
{
```

```
    int a[5], *ptr, i;
```

```
    clrscr();
```

```
    ptr = &a[0];
```

```
    cout << "Enter five array elements:";
```

```
    for (i=0; i<5; i++)
    {
```

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

```
    cout << "Array Elements are:";
```

```
    for (i=0; i<5; i++)
    {
```

```
        cout << *ptr;
```

```
        ptr++;
    }
```

```
    getch();
}
```

O/P

Enter five array elements:

10

20

30

40

50

Array Elements are: 10 20 30 40 50

* Pointers to strings:

- String is a collection of characters.
- strings are enclosed within double quotes.
- string is a array of characters.
- string terminated by a NULL character
~~as '10'.~~
- It is a character type pointer which is used with to implement string related operations.
- Syntax:

char * ptr-var-Name;

Eg:

char str[10], *ptr;

ptr = str or ptr = &str[0];

- pointer pt holds address of 0th index.

	0	1	2	3	4	5	6	7	8	9
str	H	E	L	L	O	'\0'				

- suppose string "HELLO" assign to str then memory structure look like above.
- C++ compiler automatically insert '\0' character at the end of string.

Program: calculate length of string.

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

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

```
Void main()
```

```
{
```

```
char *ptr, str[10]; int len=0;  
clrscr();
```

```
cout<<"Enter string:";  
cin>>str;
```

```
ptr = &str[0];
```

```
while (*ptr != '\0')
```

```
{
```

```
len = len + 1;
```

```
ptr++;
```

```
}
```

```
cout<<"Length of string = "<<len;
```

```
}
```

O/P

Enter string: HELLO

Length of String = 5

* string copy:

* string concatenation

* string reverse.

* string comparison

} please refer
your class Notebook
program.

* this pointer!

 x

- 'this' is a predefined keyword.
- private variables can be used directly inside a member function.
- Eg.

class ABC

{

 int a;

 public:

 void getdata()

{

 a=123; // private variable

}

 // access used directly.

};

- we can use this pointer to do this same job.

Eg.

class ABC

{

 int a;

 public:

 void getdata()

{

 this->a=123; // this pointer

}

};

- we mostly don't use this pointer explicitly.
- "this" pointer points to current calling object of method.
- one important application of this pointer is to return the object that invoked the function.

Program:

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

void main()

{

class Addition

{

int a, b, c;

public:

void getdata()

{

this → a = 10;

this → b = 20;

}

void display()

{

this → c = this → a + this → b;

cout << "In Addition = " << this → c;

}

3)

void main()

{

Addition a1;
class a1;

a1.getdata();
a1.display();

getdata();

}

OP

Addition = 30

a1 → a

(object)

| 10 |

b | 20 |

c | 30 |

* Pointers to objects:

- A pointer can point to an object created by a class.
- pointer do the same job like object.
- Consider the example:

item x;

- where:

item - is a class

x - is an object created from item class

- Similarly we can define pointer also

item *ptr;

- pointer ptr can access all the public members of class.

- pointer ptr initialized with the address of x.

Eg.

item x;

item *ptr;

ptr = &x;

- we can access member function of item class in two ways.

i) one by using dot operator:

```
x.getdata(10, 20);  
x.display();
```

ii) second, by using arrow operator:

```
ptr->getdata(10, 20);  
ptr->display();
```

program:

```
#include <iostream.h>  
#include <conio.h>  
  
class item  
{  
    int a, b;  
public:  
    void getdata(int x, int y)  
    {  
        a = x;  
        b = y;  
    }  
    void display()  
    {  
        cout << "n a = " << a;  
        cout << "n b = " << b;  
    }  
};
```

Void main()

{

item x;

Item *ptr;

clrscr();

ptr = &x;

ptr->getdata(10, 20); // Access member by
// using pointers.

ptr->display();

getch();

}

O/P

a=10

b=20

6. Polymorphism

* Polymorphism:

- Ability to take more than one form is known as polymorphism.
- polymorphism is a Greek word.
- The meaning of Poly is 'many' and morphism means 'forms'
- polymorphism means many forms.
- There are two types of polymorphism:
 - 1) compile time polymorphism
 - 2) runtime polymorphism

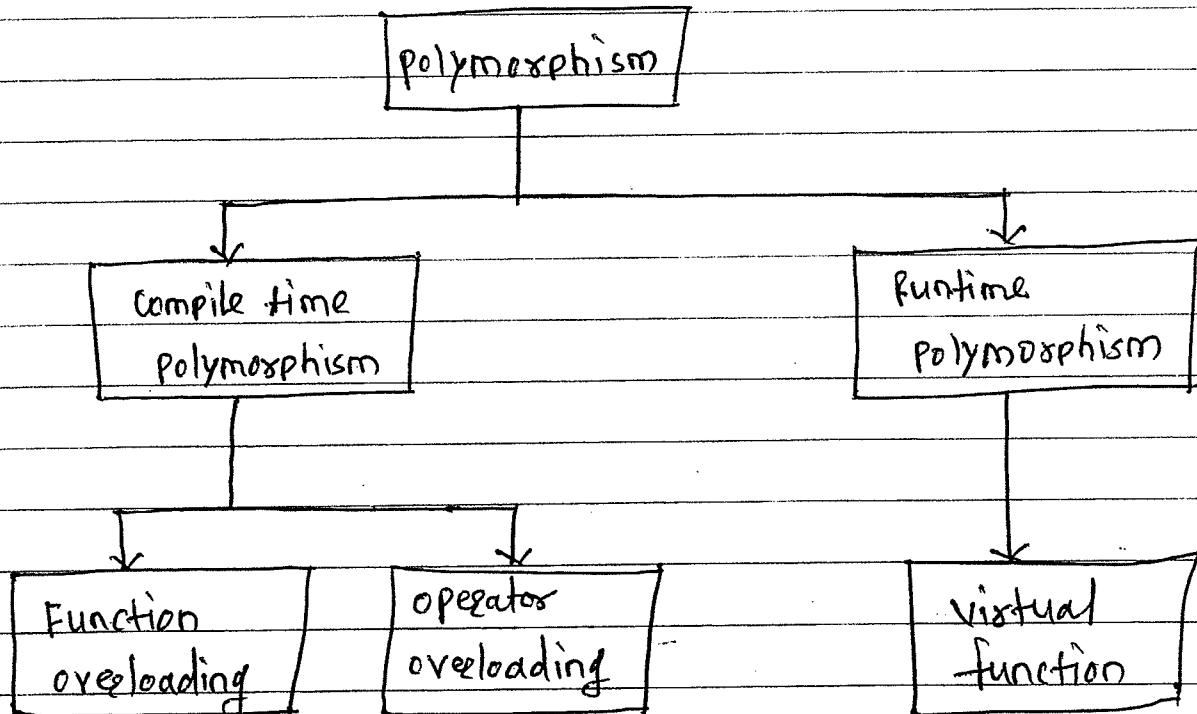


fig: types of polymorphism

- Above fig. shows compile time polymorphism contains function overloading & operator overloading.
- Runtime polymorphism contain virtual function.

* Early Binding / static Binding / Compile time polymorphism

- when function is selected for execution at the time of compilation is known as early binding or static binding.
- By default C++ follows static or early binding.
- During compilation process, C++ compiler determines the function call based on parameter & type of +
- C++ compiler invoke the correct function for each call.
- Compile time polymorphism is efficient than runtime polymorphism.
- Compile time polymorphism can be achieved in two different ways:
 - (1) Function overloading.
 - (2) operator overloading.

Function overloading :

- Overloading means to the use of same thing for different purposes.
- Function overloading means function name are same but its arguments are different.
- With the help of function overloading, many functions with same function name & different argument lists.
- When calling function is executed program controller goes to function definition.
- It matches no of arguments & type of argument.

- The function selection involves the following steps:

- 1) The compiler first tries to find an exact match of function call & function definition.
- 2) If an exact match is not found then compiler will choose function depending upon below things match.
 - char to int
 - float to double
 - int to long
- 3) When above two things are not match then compiler tries to use implicit conversion. If conversion is having multiple matches then it will throw error message.

Eg: suppose we use following two functions.

long square(long a)

double square(double x)

- A function call such as - square(10);
- This will produce error because int argument can be converted to long or double so compiler gets confuse.

Program:-

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

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

class Addition

{

int a, b, c;

public:

void getdata (int x)

{

a = x;

b = 20;

}

void getdata (int m, int n)

{

a = m;

b = n;

}

void display()

{

c = a + b;

cout << "In Addition = " << c;

}

};

void main()

{

Addition a1/a2;

clrscr();

a1.getdata(10);

a2.getdata(20, 30);

a1.display();

a2.display();

getch();

}

O/P

Addition = 30

Addition = 50

* operator overloading:

- To assign a new meaning to the existing operator is known as operator overloading.
- Operator overloading assigns special meaning to an operator.
- All C++ operators can be overloaded except below four:
 - 1) class member operators (+, -*)
 - 2) scope resolution operator (::)
 - 3) size operator (sizeof)
 - 4) conditional operator (? :)

- Operator overloading is a compile time polymorphism.

Syntax:

```
returntype className::operator op(Arg. list)
```

```
{
```

```
// function body
```

```
}
```

Where:

returntype = datatype int, float, char, etc

op = operator being overloaded.

operator = keyword

- Operator function used in class must be either member function or friend function.
- The basic difference between member function & friend function is

member function has no arguments for unary operators & one argument for binary operators
- For friend function has one argument for unary operator & two arguments for binary operator.

- overloaded function may be invoked by following expressions:

1) ^{operator} For unary we can use:

Eg. $a \text{ op}$ // a is object of op is
 $-a$ or $a -$ // operator being overloaded

2) For binary operator we can use

$a \text{ op } b$

Eg. $a + b$; // a is calling object & b is
// passed as arguments.

3) For friend function - unary operator

operator $\text{op}(a)$

Eg. operator $-(a)$

4) For friend function - binary operator

operator $\text{op}(a, b);$

Eg.

operator $+(a, b);$

* overloading unary operator:

- When operator takes one operand for doing ~~operator~~ operation, then it is called Unary operator.
- Let us consider unary minus operator.
- Unary operator takes one operand.
- This operator changes the sign of an operand.

Program:

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

class space
{
    int x1, y1, z1;
public:
    void getdata (int a, int b, int c);
    void display();
    void operator-(); //overloaded unary minus
};

void space :: getdata (int a, int b, int c)
{
    x1 = a;
    y1 = b;
    z1 = c;
}
```

Void space :: display ()

{

cout << "x1 << " " << y1 << " " << z1;

}

Void space :: operator -()

{

x1 = -x1;

y1 = -y1;

z1 = -z1;

}

Void main()

{

space s;

clsscr();

s.getdata (10, -20, 30);

cout << "s:";

s.display ();

-s; // activates operator -()

cout << "\n s:";

s.display ();

((:) getch());

}

OLP

$s: 10 -20 30$

$s: -10 20 -30$

* overloaded Binary operators:

- Binary operators which require two operand to perform operations.
- we can invoke binary operator overloaded function as follows:

$a+b;$

- object a is an invoking the function of b passed as arguments.

program:

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

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

```
class space
```

```
{
```

```
    int x1, y1, z1;
```

```
public :
```

```
    void getdata (int a, int b, int c);
```

```
    void display();
```

```
    space void operator + (space m1);
```

```
};
```

```
Void space :: getdata (int a, int b, int c)
{
```

$x_1 = a;$

$y_1 = b;$

$z_1 = c;$

}

```
Void space :: display()
```

{

~~cout << x1~~

cout << x1 << " " << y1 << " " << z1;

}

```
Space void - space :: operator + (space m1)
```

{

Space temp;

temp.x1 = x1 + m1.x1;

temp.y1 = y1 + m1.y1;

temp.z1 = z1 + m1.z1;

return (temp);

}

```
Void main()
```

{

Space s1, s2, s3;

cls();

s1.getdata(10, 20, 30);

s2.getdata(2, 4, 6);

$s_3 = s_1 + s_2 ; \quad // \text{Activated operator } +()$

`cout << "In s1 :";
s1.display();`

`cout << "In s2 :";
s2.display();`

`cout << "In s3 :";
s3.display();`

`getdata();`

3

O/P

$s_1 : 10 \ 20 \ 30$

$s_2 : 2 \ 4 \ 6$

$s_3 : 12 \ 24 \ 36$

* Rules for operator overloading :-

- 1) only existing operator can be overloaded. New operator cannot be created.
- 2) The overloaded operator must have at least one operand.
- 3) we can not change the basic meaning of an operator.
- 4) overloaded operators follow the syntax rules.
- 5) There are some operators that cannot be overloaded.

(I) `sizeof`

(II) `*` → membership operator

(III) `*` → pointer to member operator

(IV) `::` → scope resolution operator

(V) `?:` → conditional operator

- 6) we cannot use friend function to overload below operators:

(I) `=` → assignment operator

(II) `()` → parenthesis function call operator

(III) `[]` → subscripting operator

(IV) `>>` → class member access operator

* Function overriding:

- Function overriding means same function name present in base class & derived class. Then base class function overridden by derived class function.
- If same function present in base class & derived class then compiler makes decision in the following way to execute member function.
 - I) If we use base class object to invoke overridden function then base class function is executed.
 - II) If we use derived class object to invoke overridden function then derived class function is executed.
- Function overriding means function name are same in base class & derived class but different implementation.

when overriding

function

of

program:

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

```
class Base :
```

```
{
```

```
public :
```

```
void display()
```

```
{
```

```
cout << "I am from base class:";
```

```
}
```

```
};
```

```
class Derived : public Base
```

```
{
```

```
public :
```

```
void display()
```

```
{
```

```
cout << "I am from Derived class:";
```

```
}
```

```
void main()
```

```
{
```

```
Base b;
```

```
Derived d;
```

```
else
```

```
else
```

b. `display();`
 d. `display();`

`delete();`

}

O/P

I am from base class

I am from derived class

* Run time polymorphism :-

* Virtual Function :-

- Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms.

- Therefore, an essential requirement of polymorphism is the ability to refer to objects without any regard of their classes.

- This requires the use of single pointer variable to refer to the objects of different classes.

- Here, we use the pointer to base class to refer to all the derived classes objects.

- The compiler simply ignores the contents of the pointer & choose the member function

that matches the type of pointer.

- In this case polymorphism is achieved by using virtual functions.
- When we use same function name in both the class base & derived classes then function in the base class is declared as virtual function.
- When function is made virtual, C++ determine which function to use at runtime based on the type of object pointed to by the base pointer.
- Runtime polymorphism is achieved only when a virtual function is accessed through pointer to the base class.

Program:-

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

class Base
{
public:
    void display()
    {
        cout<<"Display Base:">>
    }

    virtual void show()
    {
        cout<<"Show Base:">>
    }
}
```

class Derived : public Base

{

public:

void display()

{

cout << "In display derived";

}

void show()

{

cout << "In show derived";

}

}

void main()

{

Base B;

Derived D;

Base * bptr;

(D*) bptr;

cout << "In bptr points to base;"

bptr = &B;

bptr → display();

bptr → show();

cout << "In bptr points to derived;"

bptr = &D;

bptr → display();

bptr → show();

getch();

}

O/P

bptr points to base

display base

show base

bptr points to derived

display derived

show derived

*

Rules for virtual Functions:

1) The virtual function cannot be static members.

2) The virtual functions are accessed by using object pointer.

3) It cannot be friend of another class.

4) Virtual constructor cannot be created, but virtual destructor can be created.

5) A base pointer can point to any type of derived objects, but derived pointer cannot point to the base class.

6) If a virtual function is defined in the base class, then it is not needed to be redefined in

the derived class.

- 7) In a base class virtual function must be defined, even although it may not be used.
- 8) the declaration of base class virtual function must be and all the derived class functions must be the same.

* Pure virtual functions:

- pure virtual function is a virtual function which has no body.
- pure virtual function has no body,
- programmer must add the notation = 0 for declaration of the pure virtual function within the base class.
- ~~the notation = 0 indicates virtual function is a pure virtual function~~

Syntax to declare and use:

```
class className {
public:
    virtual void virtualfunctionName() = 0 // this denotes
    int virtualFunction(); // the pure virtual function.
```

31

Program:

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

class Base
{
public:
    virtual void show() = 0 // pure virtual function
};

class derived : public Base
{
public:
    void show()
    {
        cout << "In derived show"; 
    }
};


```

Void main()

{

Base * bptr;

derived d;

clrscr();

bptr = &d;

bbt bptr->show();

getch();

}

O/P

derived show:

↳ $\text{H}_2\text{O} + \text{Na}_2\text{CO}_3 \rightarrow \text{Na}_2\text{HCO}_3 + \text{NaOH}$

↳ $\text{Na}_2\text{HCO}_3 \rightarrow \text{Na}_2\text{CO}_3 + \text{CO}_2$

↳ $\text{Na}_2\text{CO}_3 + \text{H}_2\text{O} \rightarrow \text{Na}_2\text{CO}_3 \cdot \text{H}_2\text{O}$

↳ $\text{Na}_2\text{CO}_3 \cdot \text{H}_2\text{O} \rightarrow \text{Na}_2\text{CO}_3 + \text{H}_2\text{O}$

↳ $\text{Na}_2\text{CO}_3 + \text{H}_2\text{O} \rightarrow \text{Na}_2\text{CO}_3 \cdot \text{H}_2\text{O}$

↳ $\text{Na}_2\text{CO}_3 \cdot \text{H}_2\text{O} \rightarrow \text{Na}_2\text{CO}_3 + \text{H}_2\text{O}$

↳ $\text{Na}_2\text{CO}_3 + \text{H}_2\text{O} \rightarrow \text{Na}_2\text{CO}_3 \cdot \text{H}_2\text{O}$

↳ $\text{Na}_2\text{CO}_3 \cdot \text{H}_2\text{O} \rightarrow \text{Na}_2\text{CO}_3 + \text{H}_2\text{O}$

↳ $\text{Na}_2\text{CO}_3 + \text{H}_2\text{O} \rightarrow \text{Na}_2\text{CO}_3 \cdot \text{H}_2\text{O}$