VJTech Academy

Inspiring your Success...

Degree & Diploma Coaching classes.

Object Oriented Prog. / C++

Author,

Mr. Vishal Jadhav Sir

(BE Computer, 8 Years of IT Industry experience)

2022

Venue,

VJTech Computer Academy, Maharashtra

M. 9730087674, Email- vjtechacademy@gmail.com

Date - 05/01/2017

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 language.
- 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.
-

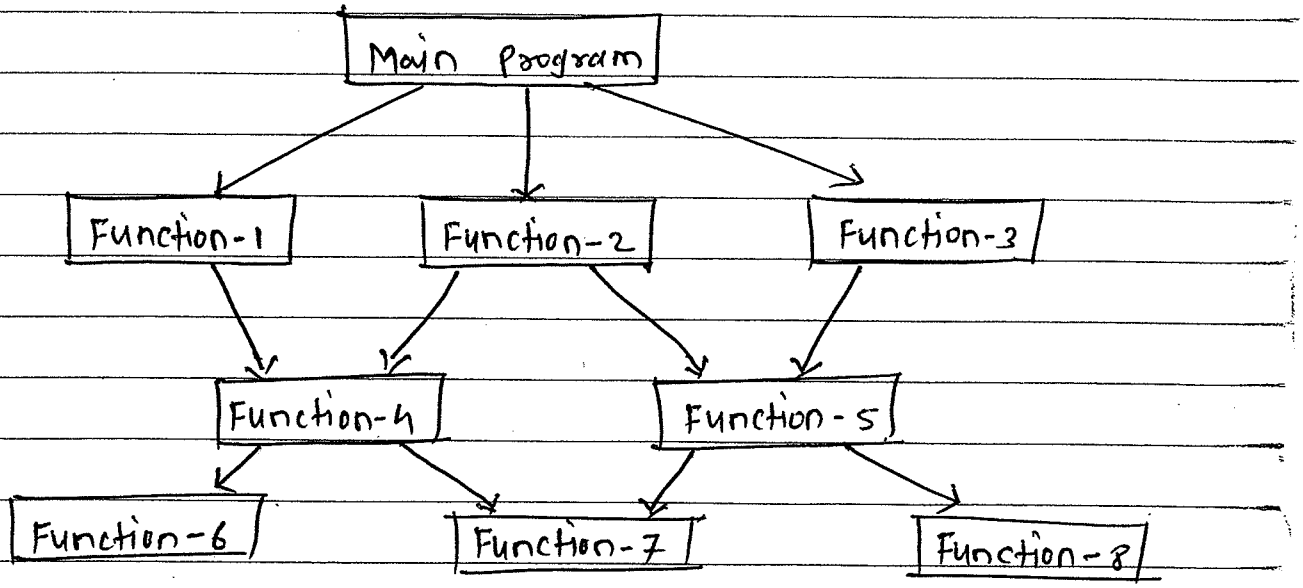


fig:- Structure of procedure oriented program

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

Characteristics of POP :-

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 pop is not hidden.
7) Modular approach programming language.

Disadvantages of POP :-

1) It does not solve real world ~~pro~~ problems very well.
2) No security for data
3)

* 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, inheritances, function overloading all these new concepts are added.
- objects are basic run time entities.
- In oop everthings are doing by using objects.
- object is a collection of data & function.

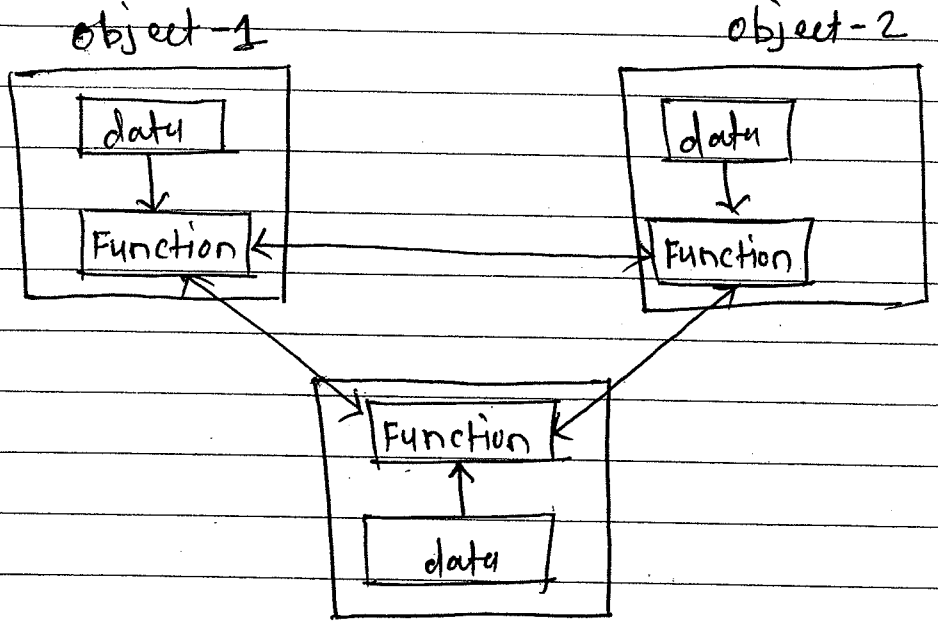


fig: structure of data & functions in oop:

- 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 is secure shows reusability feature because of inheritance. It save time in development.
ii) Data is hidden because of this programmer develope 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 functio

POP	OOP
(6) Top-down design approach	(6) Bottom-up design approach
(7) Difficult to add new data & functions	(7) easy to add new data & function.
(8) It does not solve real world problem very well	(8) It is used to solve real life problem.
(9) eg. C, FORTRAN, PASCAL	(9) eg. C++, JAVA

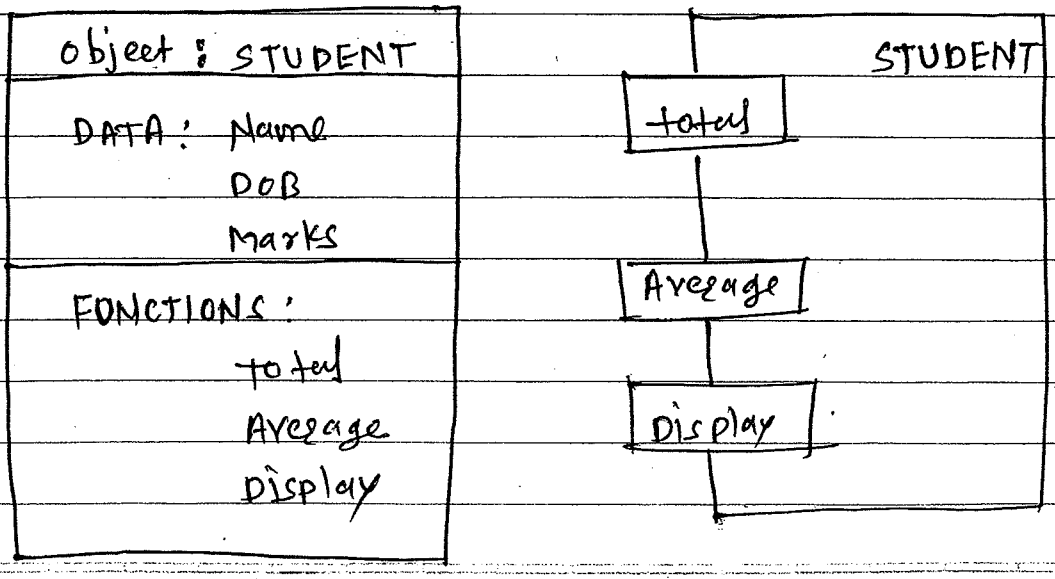
* Basic concept of object oriented programmings:

- 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.



eg.

mango, apple are objects of fruit class.

2) classes :

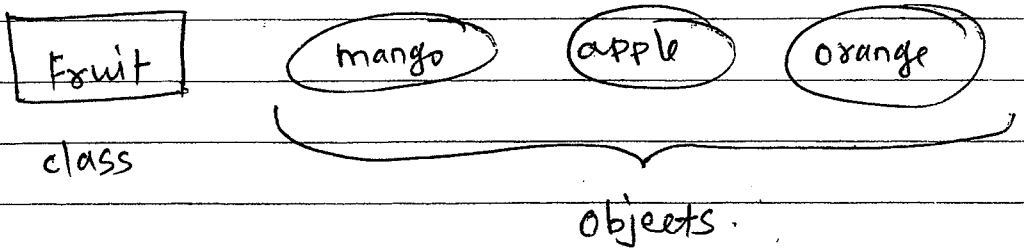
- class is a collection of objects of similiar type.
- class is a group of objects of ~~similiar~~ similar properties.
- class contains data & function.
- we can create multiple objects from class.
- classes are user defined data types.
- classes show data abstraction & data encapsulation properties.
- 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 ~~show~~ 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 provide interface betn objects & 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.
-

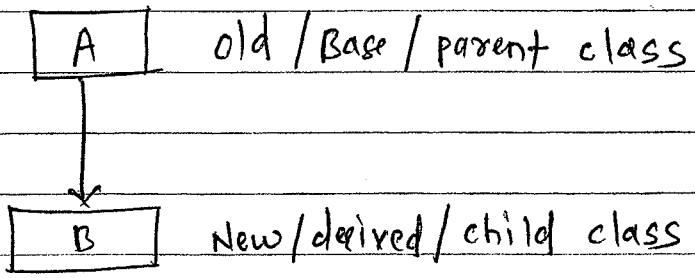


fig: Inheritance

- Above fig. shows class B is derived from base class A
- followings are the types of inheritance
1) single inheritance
2) multilevel inheritance
3) multiple inheritance
4) Hybrid inheritance
5) Hierarchical —11—

- Inheritance helps to reduce the size of programs and it saves development time.

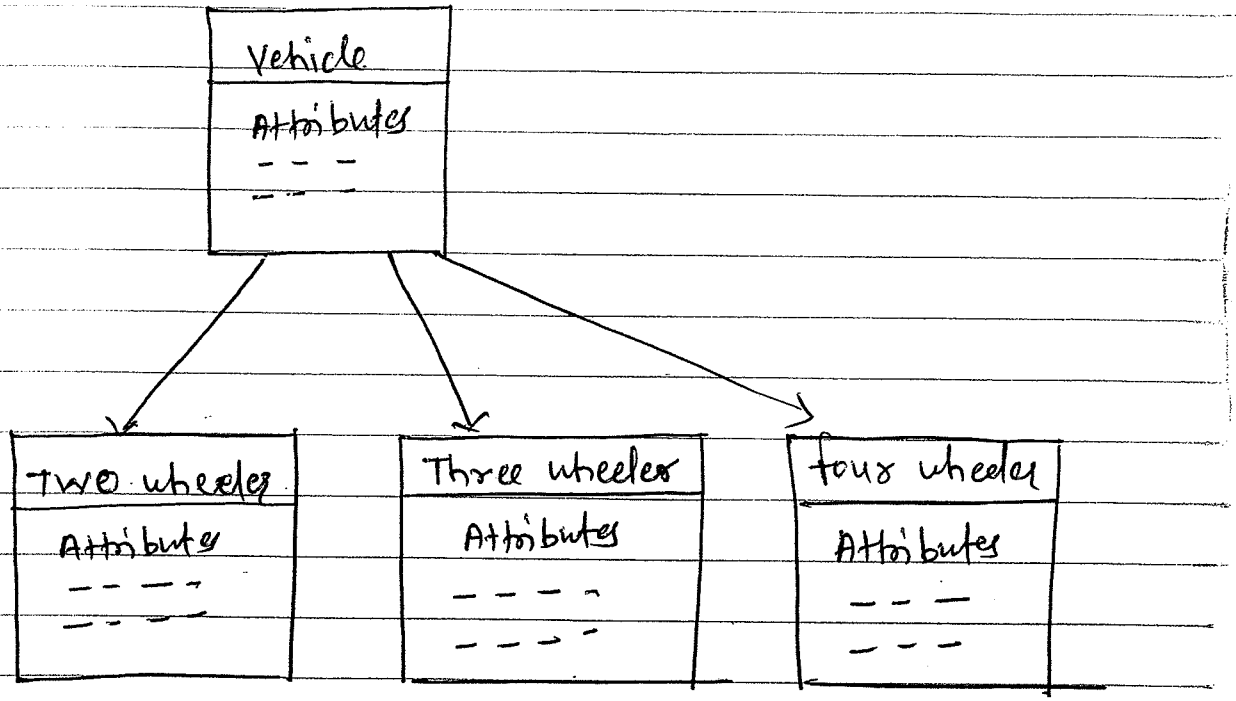


fig: Inheritance property

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 meanigs 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.

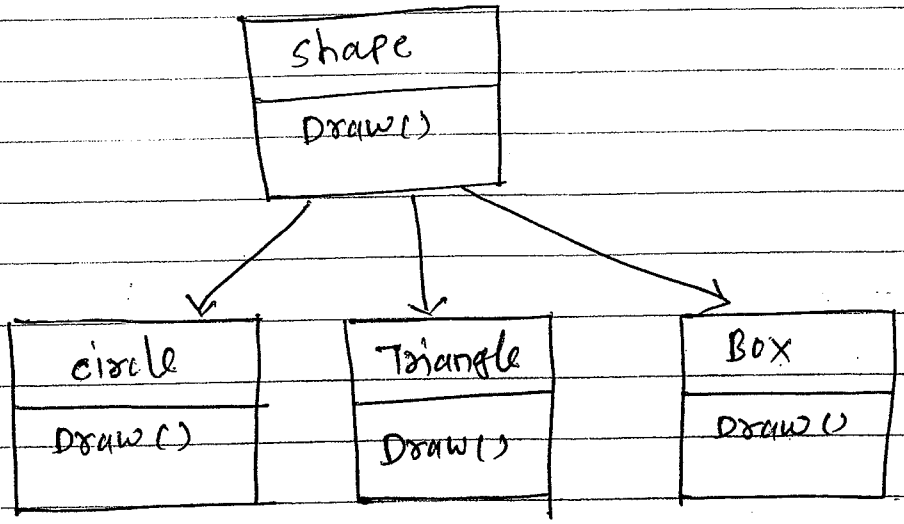


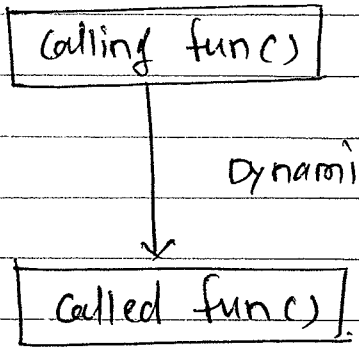
fig: polymorphism:

- 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 fun$ & called fun$.
- 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 are 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 beth 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. Identifies
3. Constants
4. strings
5. operators.

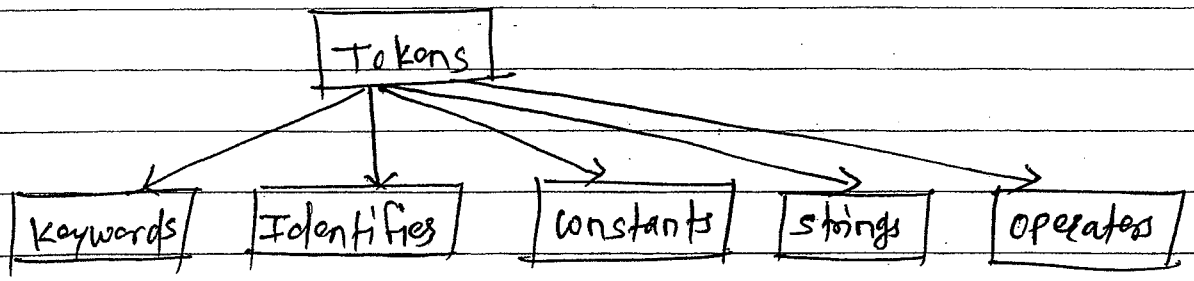


fig: tokens

* 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 indefifiers is any name of variables, function, classes etc.
- following rules are used to defining identifier

1) An identifier must be unique in program

2) It must be start from alphabets or underscore (_).
3) keywords cannot be used as identifier
4) It does not allowed any sy 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
roll$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 x const x 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<iostream.h>
#.
#include<iostream.h>
#include<conio.h>
    void main()
  {
      const int MAX = 50;
      clrscr();

Cout<<Printf" 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 variable.
- syntax:

```
datatype variableName;
```

- eg:

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

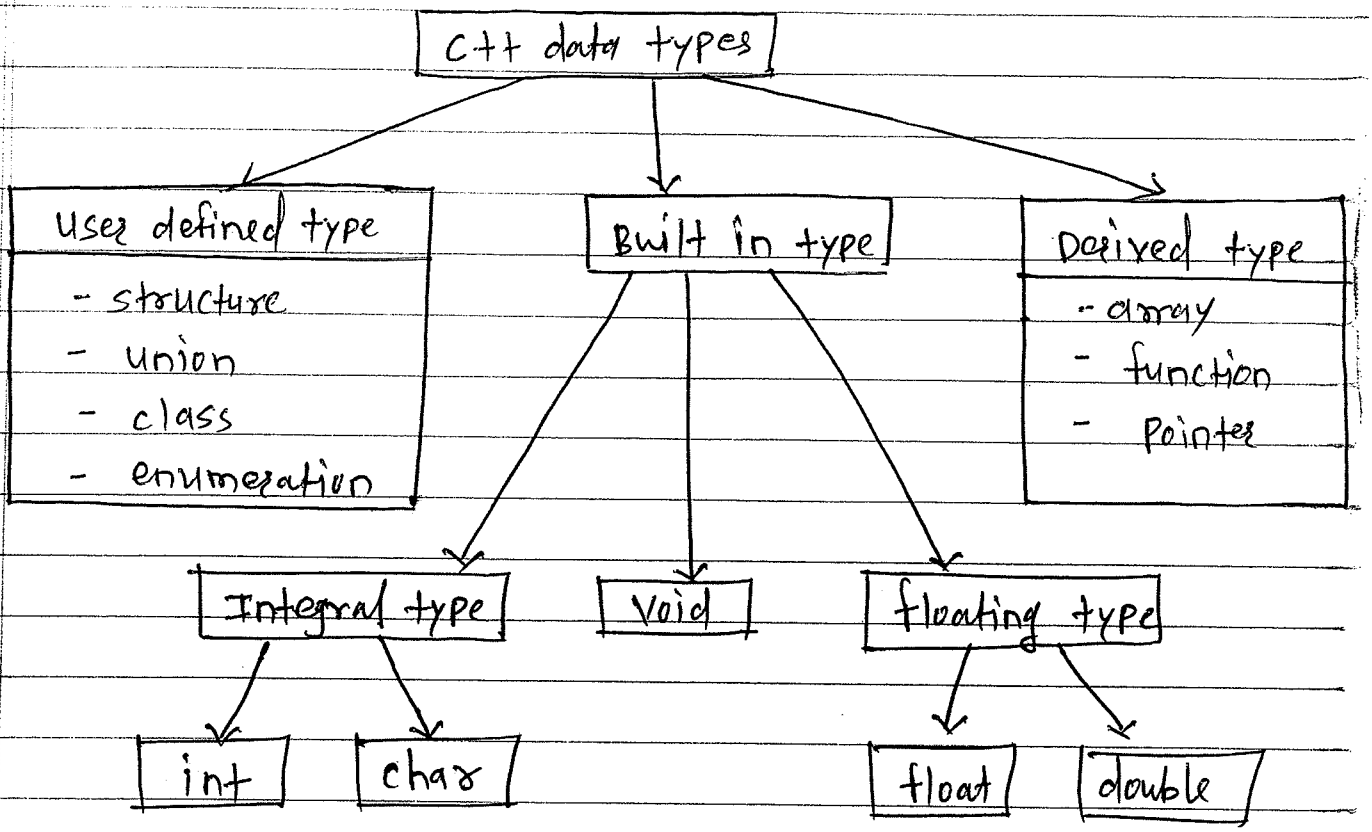


fig: C++ data types

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 '\n'
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 ~~local 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 o/p 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<<"\n Local a = "<<a;
    cout<<"\n Global a = "<<::a;

    ::a=20;
    cout<<"\n\n Local a = "<<a;
    cout<<"\n Global a "<<::a;

    getch();
}
```

o/p

```
Local a = 15 Global a = 10
Local a = 15 Global a = 20
```

* Member 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 *pvariableName = new datatype;
```

eg:

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

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

delete operator deallocates memory allocated by new.

syntax:

```
delete pvariableName;
```

eg:

```
delete p; // 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.
    getch();
}
```

o/p
string is: computer

* Structure of C++ program

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

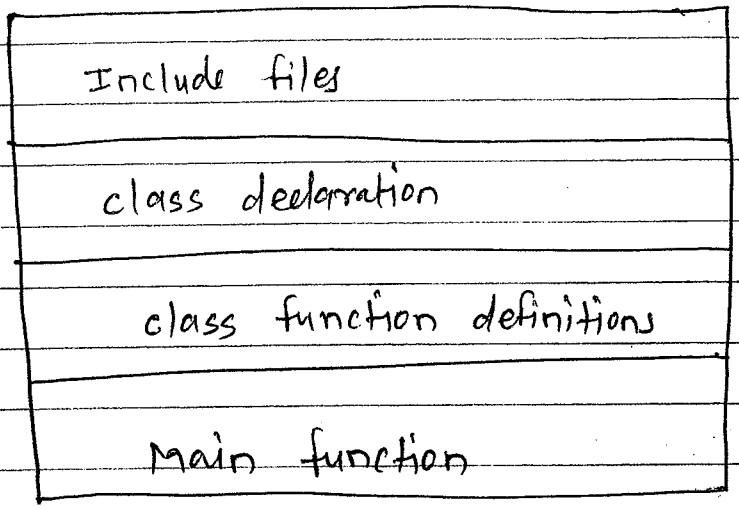


fig: structure of C++ program

- 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>
#include<conio.h>
```
} → Header files

```
void main()
{
    clrscr();
    cout<<" welcome to world of c++ "; →// c++ output
    getch();
}
```
main function

o/p
welcome to world of c++

* Input operator/stream :

- It is used to read input from uses 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>>no1;
```

- In above example program controller read value from user and stored int in no1 variable

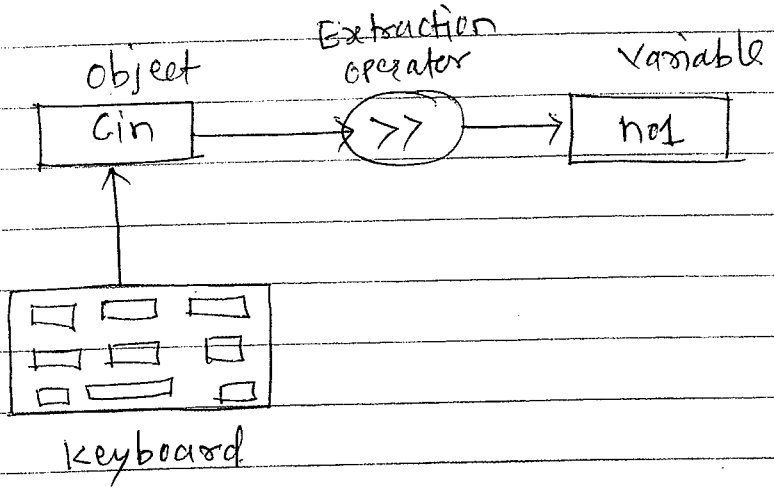


fig: Input with cin operator

eg.

1) int a;
   cin>>a;

2) float b;
   cin>>b;

3) char c;
   cin>>c;

- we can get more than one i/p using cin object

syntax:

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

eg.

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

* Output operater / stream :

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

Syntax:

```
cout << " output message " ;
```

or

```
cout << variableName;
```

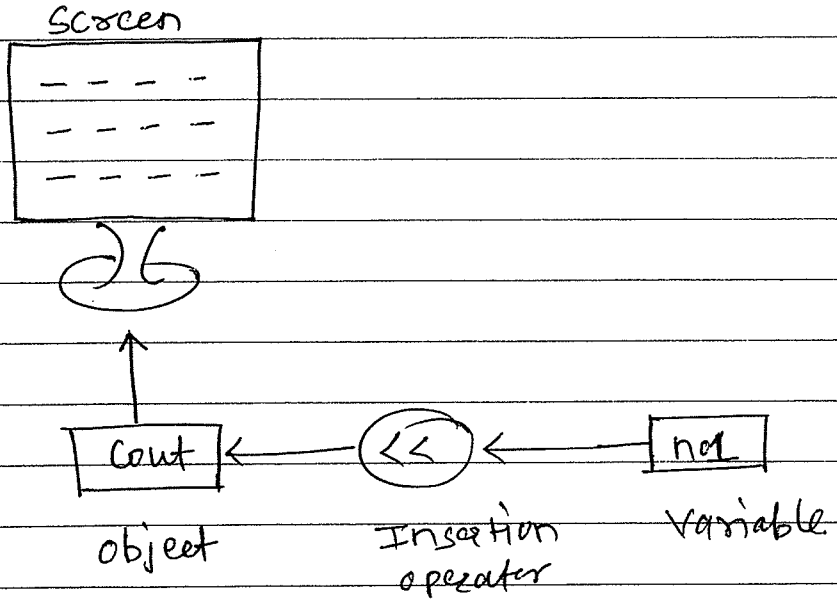


fig: output using cout operater

eg.

1) int a=10;
   cout<<a;

2) cout<<"Helloworld";

3) float b=10.11;
   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<<"\t"<<b<<"\t"<<c;
```

2. Objects and classes

DATE

* 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
};
```

eg.

```
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.

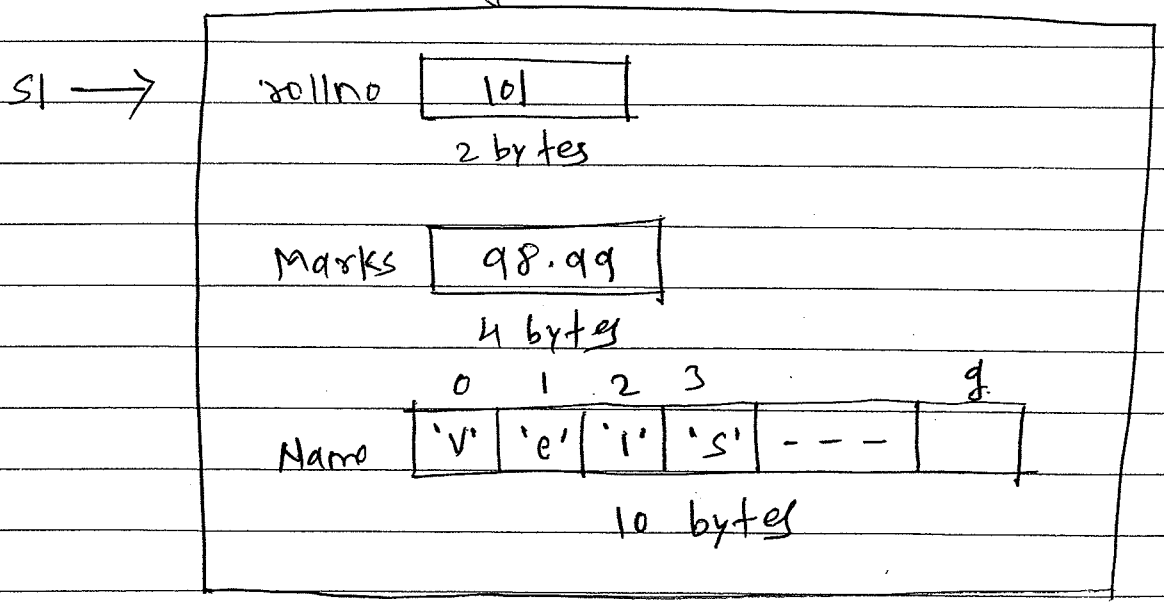


fig: Memory Representation of structure variable

program:

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

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

void main()
{
    struct student s1;
    clrscr();

    cout<<"\n Enter student rollno:";
    cin>>s1.rollno;
    cout<<"\n Enter student marks:";
    cin>>s1.marks;
    cout<<"\n Enter 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();
}
```

o/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	class
i)	It is a collection of different types of data	i) class is a collection of data and function.
ii)	'struct' keyword is used.	ii) 'class' keyword is used.
iii)	structure variables can be created	iii) class variables called as objects.
iv)	Functions are not present in structure.	iv) Functions are present in structure.
v)	No private & public member declaration	v) Members can be declared as private, public & protected.
vi)	No data hiding	vi) Data hiding
vii)	syntax:	vii) syntax:

* classes:

- clas is one of the important feature of C++.
- class is a collection of similiar types of objects.
- class contain data & function.
- class is a ~~used~~ 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 << "\n Enter rollno :";
            cin >> rollno;
            cout << "\n Enter Name :";
            cin >> name;
            cout << "\n Enter Marks :";
            cin >> marks;
        }
        void display ()
        {
            cout << "\n Roll NO : " << rollno;
            cout << "\n Name : " << name;
            cout << "\n 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.

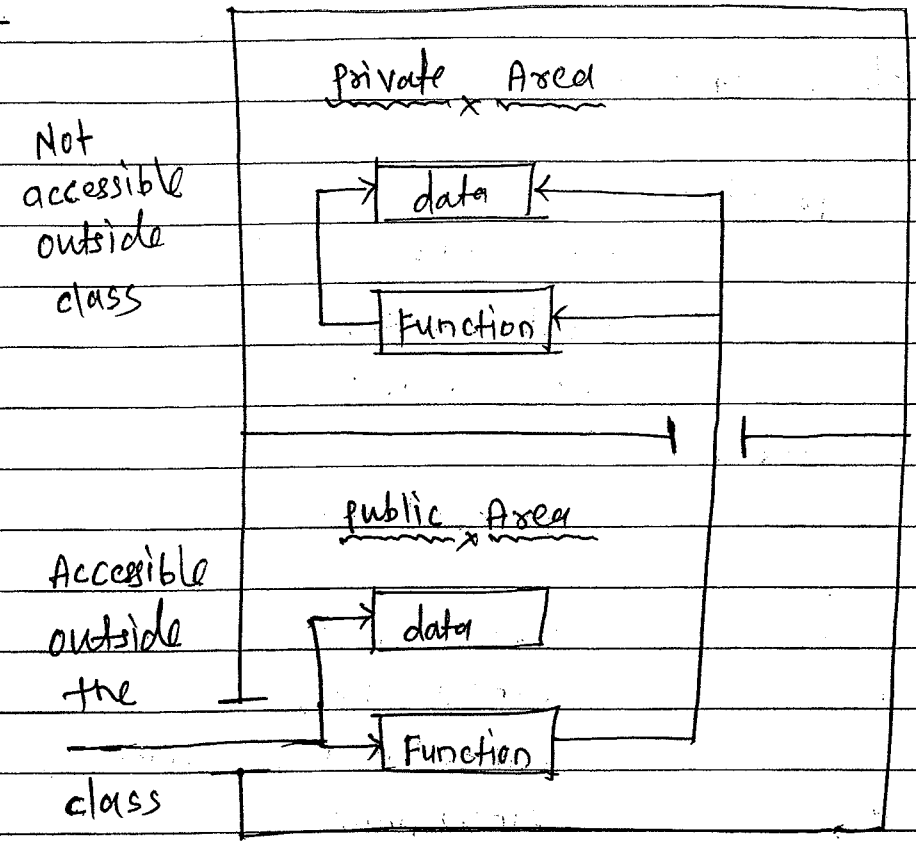


fig: Representation of classes:

* 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;   // 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 usi~~ through the member functions of that class.
- Syntax:

```
objectName . dataMember;
        or
objectName . member function (Argument);
```

Eg.

```
s1.getdata();
s1.display();
```

- The 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^n definition
        {
            a=10;
            b=5;
        }
        void display() // inside fun^n definition.
        {
            c=a+b;
            cout<<"\n Addition = "<<c;
        }
};
```

② Outside the class definition:

- Member function declared inside the class and it's definition written outside the class.
- while writting function definition outside the class then we need to use className and slope resolution operator (::).

Syntax:

```
returntype className :: function_Name(arg. list)
{
    // Function body;
}
```

Eg.

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

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

```
Void Addition::display()
{
    c = a + b;
    cout << " In Addition = " << c;
}

Void main()
{
    Addition a1;
    clrscr();

    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:

```
className  object_Name[size];
```

Eg:

```
class Addition
{
    private:
        int a,b;
    public:
        void getdata()
        {
            cout<<"\n Enter two numbers:";
            cin>>a>>b;
        }
        void display()
        {
            cout<<"\n Addition = "<<a+b;
        }
};
```

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

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

    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 allocates 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 S1, S2;

- In above example S1 & S2 both are objects.
- Each objects has two data members x & y.
- so total 4 bytes memory allocated for each objects.

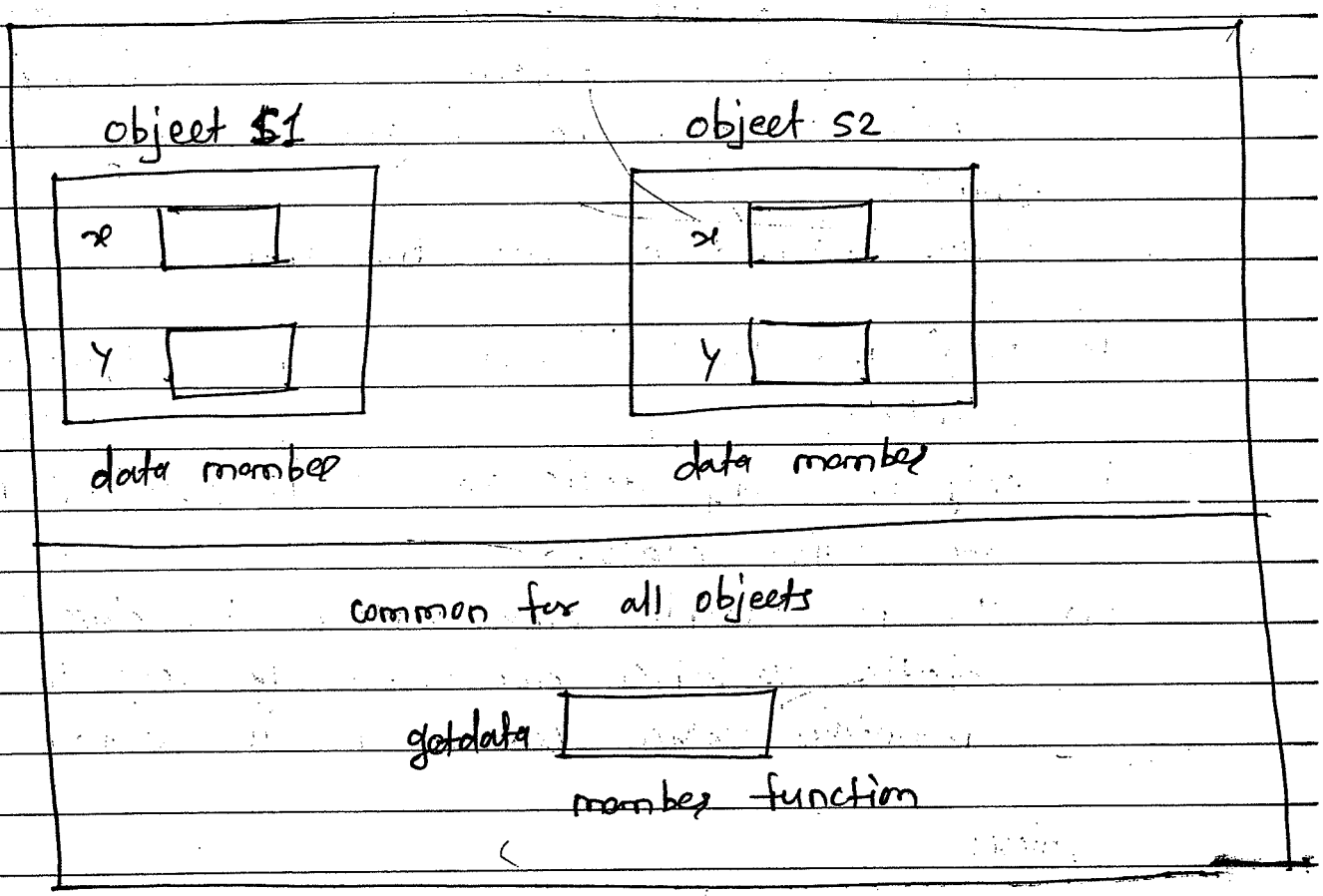


fig: objects in memory

- 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 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 copy of 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.

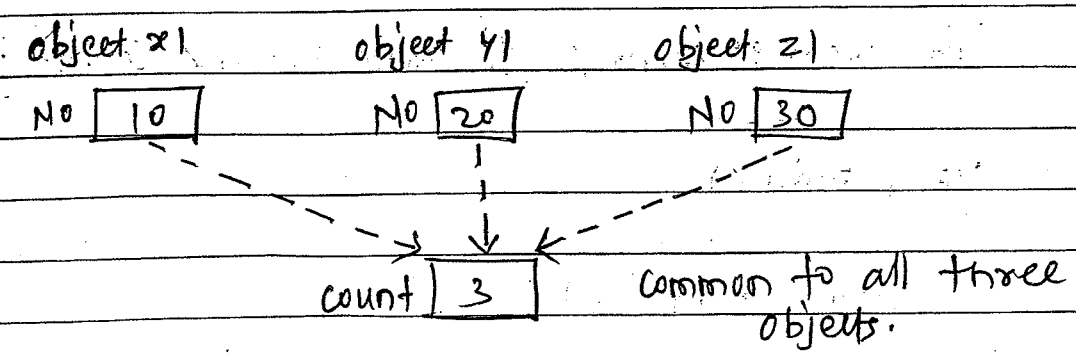
program:

```
#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 << "\n count = " << count;
          }
};
int item :: count; // static variable count is defined.

Void main()
{
   item x1, y1, z1;
```

```
clrscr();

x1.display();
y1.display();
z1.display();

x1.getdata(10);
y1.getdata(20);
z1.getdata(30);

x1.display();
y1.display();
z1.display();

getch();
}
```

o/p

```
Count = 0
Count = 0
Count = 0
Count = 3
Count = 3
Count = 3
```



* Static Member Function :

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

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

- Characteristics of a static member function:

i) A static member functions can access only other static members.

ii) static member function can be called using class name.

```
className :: static_function_Name
```

iii) They may not be virtual.

iv) They do not have 'this' pointer.

v) There can not 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^n
          {
             cout<<"\n count="<<count;
          }
};
int item :: count;

void main()
 {
    item x1, y1, z1;
    clrscr();
    x1.display();
    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 dcln.

- No need to use class name & scope resolution operator while writting friend funn 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<<"\n 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 it's 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 should be present in public section
iv) It called automatically when object is created.
v) Constructor can not be inherited.
vi) Constracter can not be virtual.
vii) It is used to initialize objeet of class.
viii) Constructor is a special member function of class.

* Default constructor :

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

syntax:

```
class class_Name
{
  - - -
  - - -
public:
  class_Name() // default constructer
  {
    - - -
    - - -
  }
};
```

Eg.

```
class Addition
{
    int a, b; c;
    public:
        Addition()  //default constructor
        {
            a=10;
            b=5;
        }
        ---
        ---
}
```

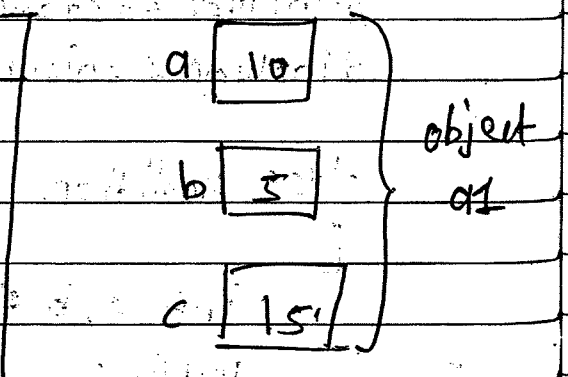
- In above example Addition() is default constructor.

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 << "\n Addition = " << c;
}

};

Void main()
{
    Addition a1;
    clrscr();

    a1.display();

    getch();
}
```

o/p
Addition = 15



* **Parameterized Constructor :**

- parameterized constructer is one of the type of constructer.
- when constructer takes some argument than it is called parameterized constructer.
- A constructer accept arguments is called as parameterized constructer.
- constructer name & class name both are same.
- There is no any return type for constructer.

Syntax :

```
class class_Name
{
   ----
   public:
      class_Name (parameter lists)
      {
         ---
         ---
      }
      ---
};
```

Eg.

```
class Addition
{
   int a, b, c;
   public:
      Addition (int x, int y)  // parameterized constr
      { 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<<"\n Addition ="<<c;
      }
};
```

```
Void main()
{
    Addition a1(10,5); // pass arguments to constructor.
    clrscr()

    a1.display();

    getch();
}
```

o/p
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 it's own class as a parameter is called as a copy constructor.
- When constructor takes reference of 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:
    className (className &objName)
    {
      ---
    }
  ---
};
```

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

    clrscr();

    code
    Cout<<"\n object c1 ID: ";
     c1.display();

    Cout<<"\n object c2 ID:";
      c2.display();

    getch();
}
```

o/p
object c1 ID: 10 object c2 ID: 10

* Constructor overloading:

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

Eg.

```
class code
{
    int id;
    public:
        code()          // default constructer
        {
            id = 100;
        }
        code (int x)    // parameterized constructer
        {
            id = x;
        }
        code (code &m)  // copy constructer
        {
            id = m.id;
        }
```

```
Void display()
{
    cout<<id;
}
};

Void main()
{
    code c1;

    code c2(200);

    code c3(c2);

    cout<<"\n object c1 id:";
    c1.display();

    cout<<"\n object c2 id:";
    c2.display();

    cout<<"\n object c3 id:";
    c3.display();

    getch();
}
```

o/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 overloading.

* Constructors with Default Arguments:

- It is possible to define constructer with default argument.
- Eg:
  constructer Addition() can be declared as follow.

```
Addition (int x, int y=0);
```

- The default value of the argument y is zero.
- if we write call 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 overcide.

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<<"\n Addition=" << c;
}
getch();
};
void main()
{
   Addition a1;
   clrscr();
   a1.display();
   getch();
}
```

o/p

Addition = 15

* Destructer :

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

- Destructer also present in public section of class.
- Destructer can not be inherited.
- Syntax:

```
className
{
- - -
public:
```

```
~className()  // destructer
{
   ---
   ---
}
---
};
```

Eg:

```
int count=0;
class Alpha
{
   public:
      Alpha()  // constructer
      {
         count++;
         cout<<"\n No of object created: "<<count;
      }

      ~Alpha()  // Destructer
      {
         cout<<"\n No of object destroyed: "<<count;
         count--;
      }

};
```

```
Void main()
{
    clrscr();

    Alpha a1, a2;
    {
        Alpha a3;
    }
}   Alpha a4;

    getch();
}
```

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
```

* Difference betn Constructer and Destructer

	Constructer	Destructer
1)	Constructer is used to create objects	1) Destructer is used to destroy the objects.
2)	It's name is same as class name	2) It name is same as class nam but preceded by tilde (~).
3)	Constructor can take arguments.	3) Destructer does not take any arguments.
4)	Constructer is involced when object is created.	4) It is involced by copiler when program reaches closing curly bracket.

5)	Eg.
	Addition()
	{
	- -
	- -
	}
		Eg:
		~Addition()
		{
		- - -
		- - -
		}

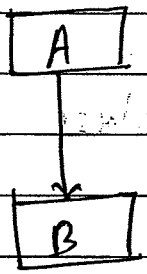
4. Inheritance

PAGE NO.
DATE 18 03 2017

* Inheritance:

- Inheritance is a one of the major important properties of OOP.
- To create new class by using the properties to 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.



old / Base / super / parent class

New / derived / child class

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
{
    // members of derived class
}
```

- 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 modifiess - 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.
};
```

- 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 type of inheritance.
-

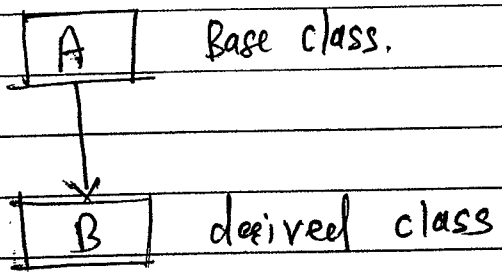


fig: single inheritance

- In above figure, we can see et. 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<<"\n Entee two values:";
           cin>>a>>b;
        }
};

class B: public A
{
    protected:
        int c;
    public:
        void display()
        {
            c=a+b;
            cout<<"\n Addition="<<c;
        }
};

Void main()
{
    B obj;
    clrscr();
    obj.getdata();
    obj.display(); getch();
}
```

o/p

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.

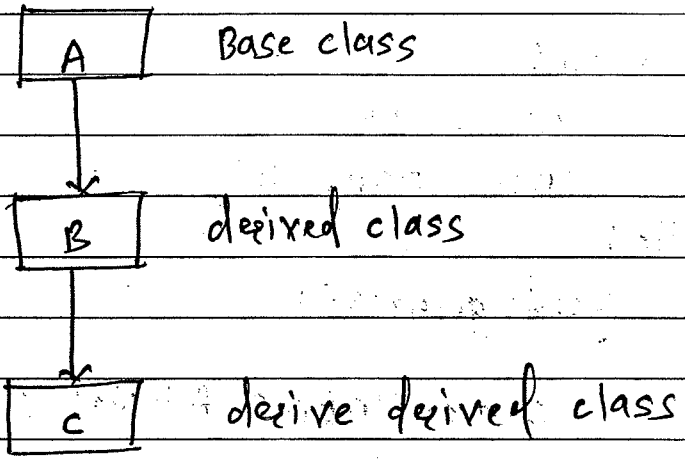


fig: multilevel inheritance

- 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<<"\n Enter Roll No: ";
            cin>>rollno;
            cout<<"\n Enter Name: ";
            cin>>name;
        }
        void putdata()
        {
            cout<<"\n Roll No: "<<rollno;
            cout<<"\n Name: "<<name;
        }
};

class Test : public student
{
    protected:
        int mark1, mark2;
    public:
```

```
void getdata1()
{
    cout<<"\n Enter mark1 : ";
    cin>>mark1;
    cout<<"\n 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;
    clrscr();
```

```
R1.getdata1();
R1.getdata2();
R1.putdata1();
R1.putdata2();
R1.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 deiving 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.

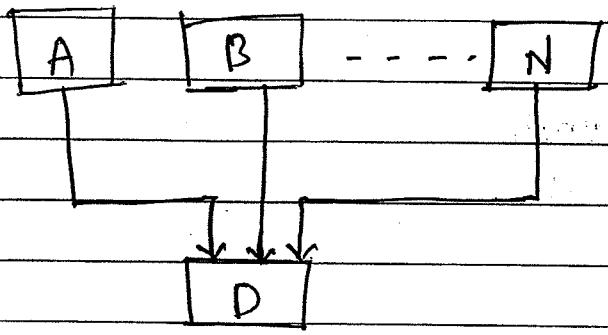


fig: multiple inheritance

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

Syntax:

```
class deived_class :visibility_mode Base1, visibility_mode Base2
{
    // Members of deived class
};
```

- Base classes are separeted 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:

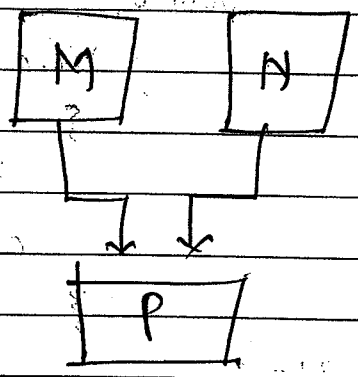
```
#include<iostream.h>
#include<conio.h>

class M
{
   protected:
         int a;
   public:
         void get_a()
         {
            cout<<"\n Enter value of a: ";
            cin>>a;
         }
         Void put_a()
         {
            cout<<"\n a="<<a;
         }
};
```

```
class N
{
  protected:
      int b;
  public:
      void get_b()
      {
        cout<<"\n Enter 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;
    clrscr();

    P1.get_a();

    P1.get_b();

    P1.disp

    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.

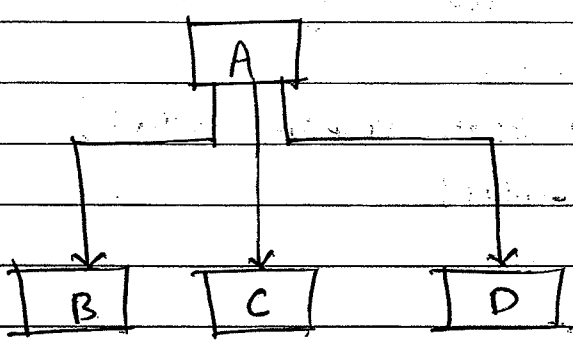


fig: Hierarchical Inheritance

- 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<<"\n Enter employee id:";
        cin>> emp_id;
        cout<<"\n Enter employee Name:";
        cin>> emp_name;
    }

    Void putdata()
    {
        cout<<"\n Emp id: "<< emp_id;
        cout<<"\n Emp Name: "<< emp_name;
    }
};

class worker : public Employee
{
    protected:
        int overtime_sal;
    public:
        void getdata1()
        {
            cout<<"\n Enter overtime salary:";
            cin>> overtime_sal;
        }
        Void putdata1()
        {
```

```
    cout<<"\n overtime salary:"<< overtime_sal;
  }
};

class Manager : public Employee
{
  protected:
      int Additional_allowance;
  public:
      void getdata2()
      {
        cout<<"\n Enter Additional Allwance:";
        cin>>Additional_allowance;
      }
    void putdata2()
      {
        cout<<"\n Additional Allwance:"<<
          Additional_allowance;
      }
};
Void main()
{
    Worker W1;
    Manager M1;
    clrscr();

    cout<<"\n Enter Worker details:";
```

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

cout<<"\n Enter workes manages details:";

M1.getdata();
M1.getdata2();

cout<<"\n *** Worker information ***";
 w1.putdata();
 w1.putdata1();

cout<<"\n *** Manager information ***";

 M1.putdata();
 M1.putdata2();

 getch();
}
```

O/P

```
Enter workes details:
Enter employee id: 14864
Enter employee Name: rishal
Enter overtime salary: 800
Enter Manager details:
Enter employee id: 32956
Enter employee Name: Nilesh
Additional allowance: 1500
```

*** worker information ***
Emp id: 148464
Emp Name: Vishal
overtime salary: 800
*** manager information ***
Emp id: 32956
Emp Name: Nilesh
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 inheritence to solve problem.

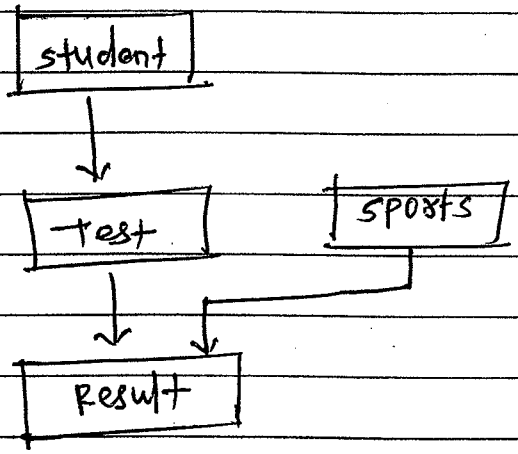


fig: Hybrid Inheritance

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

Program:

#include <stdio.

program:

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

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<<"\n Enter Mark2:";
    cin>> mark2;
  }

  Void putdata2()
  {
      cout<<"\n Mark1 = "<<mark1;
      cout<<"\n Mark2 = "<<mark2;
  }

};

class sports
 {
   protected:
       float sportwt;
   public:
       void getdata2()
       {
           cout<<"\n Enter sport weight :";
           cin>> sportwt;
       }
       Void putdata2()
       {
           cout<<"\n sport weight = "<< sportwt;
       }

};
```

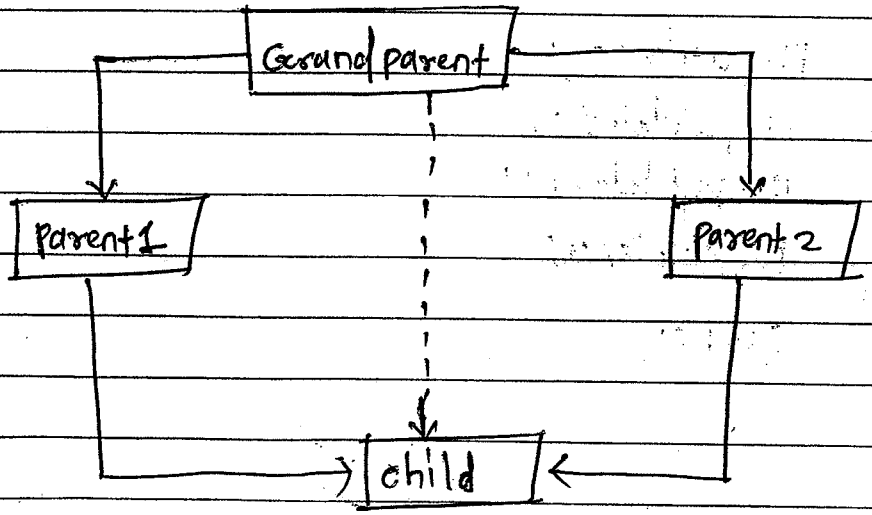
```
class Result : public Test, public sports.
{
    protected:
        int total;
    public:
        void display()
        {
            total = mark1 + mark2;
            cout<<"\n total = "<<total;
        }
};
void main()
{
    Result R1;
    clrscr();

    R1.getdata();
    R1.getdata1();
    R1.getdata2();

    R1.putdata();
    R1.putdata1();
    R1.putdata2();
    R1.display();

    getch();
}
```

o/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 :

- 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 parent1 & parent2 which is derived from common base class Grandparent.
- child class inherits the properties of Grandparent class from parent1 & parent2.
- 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<<"\n Enter RollNo:";
            cin>>rollno;
        }

        void putdata()
        {
            cout<<"\n RollNo:"<<rollno;
        }
};
```

```
class test : public virtual student
{
    protected:
        int mark1, mark2;
    public:
        void getdata1()
        {
            cout<<"\n Enter Mark1 : ";
            cin>>mark1;
            cout<<"\n Enter Mark2 : ";
            cin>>mark2;
        }
        Void putdata1()
        {
            cout<<"\n Mark1 = "<<mark1;
            cout<<"\n mark2 = "<<mark2;
        }
};

class sports : virtual public student
{
    protected:
        int sportwt;
    public:
        void getdata2()
        {
            cout<<"\n Enter sport weight : ";
            cin>>sportwt;
        }
```

```
Void putdata2()
{
    cout<<"\n sport weight="<<sportwt;
}
};

class Result : public test, public sports
{
    protected:
        int total;
    public:
        void display()
        {
            total = mark1 + mark2;
            cout<<"\n Total = "<< total;
        }
};
Void main()
{
    Result R1;
    clrscr();

    R1.getdata();
    R1.getdata1();
    R1.getdata2();

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

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

getch();
}
```

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):
        Baseclass1 (Arglist1),
        Baseclass2 (Arglist2),
          - - -
          - - -
        BaseClassN (ArglistN)
    {
        // Body of derived class constructor
    }
```

- 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 <conio.h>

class Alpha
{
    int x;
  public:
    Alpha(int i)
    {
        x=i;
    }
    void show_x()
    {
        cout<<"\n x = "<<x;
    }
};

class Beta
{
    int y;
  public:
```

```
Beta (int j)
{
    y=j;
}
void show-y()
{
    cout << "\n 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 << "\n m=" << m;
        cout << "\n n=" << n;
    }
};
```

```
Void main()
{
    Gamma g(10,20,30,40);
    clrscr();

    g.show-x();
    g.show-Y();
    g.show-mn();
    getch();
}
```

o/p
x = 10 Y = 20 m = 30 n = 40

5. Pointer in C++

PAGE NO.
DATE

* **Pointer :**

- 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, character & 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;

printf("\n a=%d",&a); cout<<"\n a="<<a;
printf("\n *p=%d",*p); cout<<"\n b="<<b;

   getch();
}
```

o/p
a = 10 *p = 10

Advantages of pointer:

1) It allows to pass variables, arrays, functions, string & structures as function arguments.

2) It supports dynamic memory allocation & deallocation of memory.

3) pointer improve efficiency of program

4) Using pointer, variables can be swapped without physically moving them.

5) It allows to establish link between data elements of object.

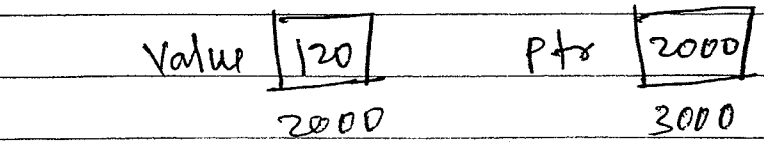
* Pointer Arithmetic :

- pointer is a variable which holds the memory address of another variable.
- some arithmetic operations can be performed with pointers.
- C++ support four arithmetic operators:

i) Addition (+)
ii) substraction (-)
iii) Incrementation (++)
iv) Decrementation (--)

Eg:

```
int value, *ptr;
value = 120;

ptr = &value;

cout printf("In Before incrementation" << ptr;
printf ("Address of

printf ("%d", ptr);

ptr++;
cout printf("In After incrementation: " << ptr;
Printf("%d", Ptr); cout << ptr;
```



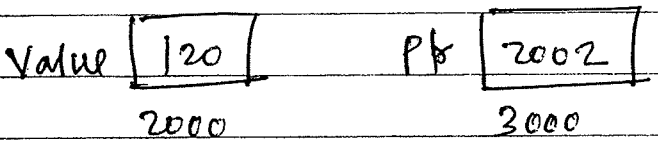
- Pointer variable ptr originally holds 2000 memory address of variable value.
- But after incrementation, it contains 2002 address.
- Because ++ operator increase the address by size of data types.

program:

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

Void main()
{
    int value, *ptr;
    clrscr();
    value = 120;

    ptr = &value;

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

    ptr++;

    cout<<"\n Memory address after incrementation:";
    cout<<ptr;

    getch();
}
```

o/p

Memory address before incrementation : 2000
memory address after incrementation : 2002



- Initially ptr 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 substract type float or double.

* Pointers to Arrays:

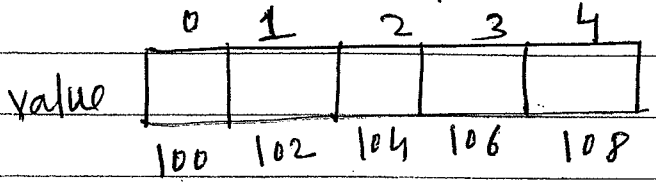
- We can use pointer concept with array.
- Consider the declaration,

```
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 points 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<<"\n Enter five array elements:";

    for(i=0; i<5; i++)
     {
        cin>>a[i];

     }


    cout<<"\n 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 or '\0'.
- It is a character type pointer which is used with to implement string related operations.
- Syntax:

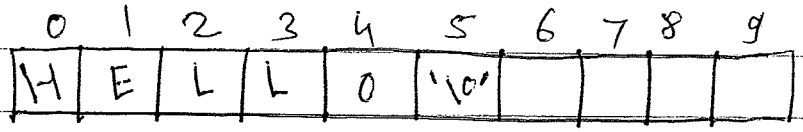
```
char * ptr_var_Name;
```

Eg:

```
char str[10], *ptr;
```

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

- pointer ptr holds address of 0th index.



- 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<<"\n Enter string: ";
     cin>> str;

     ptr = &str[0];

     while (*ptr != '\0')
      {
          len = len + 1;
          ptr++;
      }

     cout<<"\n 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:

- '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.

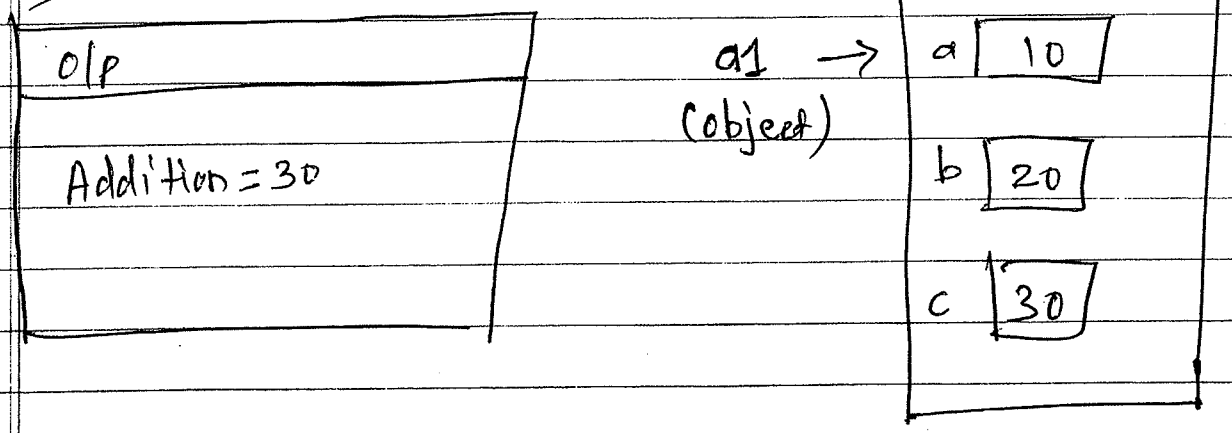
```
class ABC
{
    int a;
  public:
    void getdata()
    {
        this->a=123;   // this pointer
    }
    ---
};
```

- we mostly donot 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;
      }
```

```
};

Void main()
{
    Addition a1;
    clrscr();

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

    getch();
}
```



* 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();
```

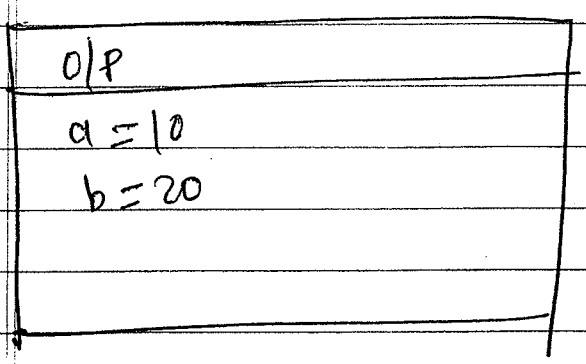
ii) second, by using arrow operator:

```
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 pointer.
    ptr->display();

    getch();
}
```



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.

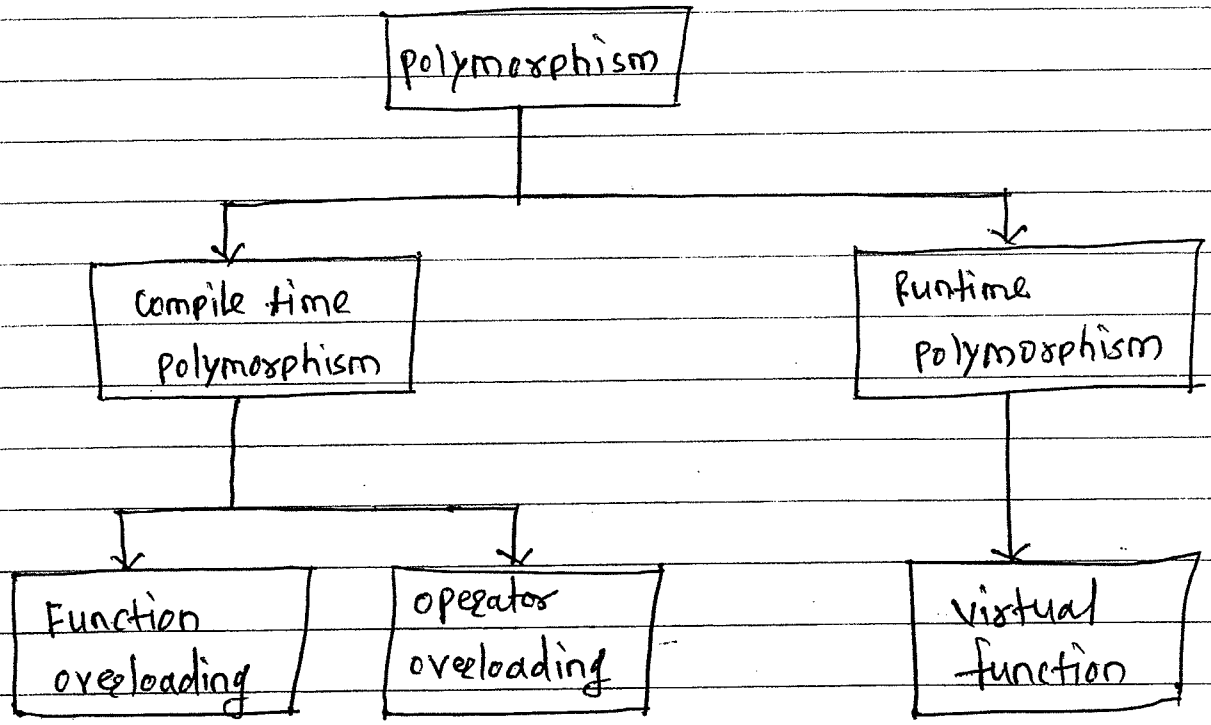


fig: types of polymorphism

- Above fig. shows compile time polymorphism containg 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 it.
- 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 it's 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 with 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<<"\n 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 assign 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 operator
- For friend function has one argument for unary operator & two arguments for binary operator.
- overloaded function may be invoked by following expressions:

1) For unary operator we can use:
   op a or a op // a is object & op is
   Eg. // operator being overloaded
   -a or a-

2) For binary operator we can use
   a op b
   Eg. a+b; // a is calling object & b is
   // passed as arguments.

3) For friend function - unary operator
   operator op(a)
   Eg. operator -(a)

4) For friend function - binary operator
   operator op(a,b);
   Eg.
   operator +(a,b);

* overloading unary operater:

- when operater takes one operand for doing operation. then it is called unary operater.
- Let us consider unary minus operater.
- Unary operater takes one operand.
- This operater 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::operater-()
{
    x1=-x1;
    y1=-y1;
    z1=-z1;
}
void main()
{
    space s;
    clrscr();
    s.getdata(10,-20,30);
    cout<<"s:";
    s.display();
    -s;    // activates operater-()
    cout<<"\n s:";
    s.display();
    getch();
}
```

o/p
S: 10 -20 30 S: -10 20 -30

* overloaded Binary operator:

- 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 & 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 &);
};
```

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

Void space :: display()
{
    Cout << x1 << " " << y1 << " " << z1;
}

space 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;
    clrscr();
    S1.getdata (10, 20, 30);
    S2.getdata (2, 4, 6);
```

```
S3 = S1 + S2 ; // Activates operator+()

cout << "\n S1 : ";
   S1.display();

cout << "\n S2 : ";
   S2.display();

cout << "\n S3 : "
     S3.display();

   getch();
}
```

O/P

```
S1 : 10 20 30
S2 : 2 4 6
S3 : 12 24 36
```

* Rules for operator overloading :-

1) only existing operator can be overloaded. New operator cannot created.
2) The overloaded operater must have at least one operand.
3) we can not change the basic meaning of an operator.
4) overloaded operater follow the syntax rules.
5) There are some operaters that cannot be overloaded.
   (I) sizeof
   (II) . → memership operater
   (III) .* → pointer to member operater
   (IV) :: → scope resolution operater
   (V) ?: → conditional operater.
6) we cannot use friend function to overload below operaters:
   (I) = → assignment operater
   (II) () → parenthesis function call operater
   (III) [] → subscripting operater
   (IV) → → class member access operater.

* 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 overridden function

↓

of

↓

Program:

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

class Base
{
    public:
        void display()
        {
            cout<<"\n I am from base class:";
        }
};
class Derived:public Base
{
    public:
        void display()
        {
            cout<<"\n I am from Derived class:";
        }
};
void main()
{
    Base b;
    Derived d;
    clrscr();
```

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

    getch();
}
```

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 refers to objects without any regard of their classes.
- This require 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<<"\n Display Base:";
        }
    virtual void show()
        {
            cout<<"\n show base:";
        }
};
```

```
class Derived : public Base
{
    public:
        Void display()
        {
            cout<<"\n Display derived";
        }
        Void show()
        {
            cout<<"\n show derived";
        }
};
Void main()
{
    Base B;
    Derived D;
    Base *bptr;
    clrscr();

    Cout<<"\n bptr points to base:";

      bptr = &B;
      bptr -> display();
      bptr -> show();

    cout<<"\n 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 can be friend of another class.
4) Virtual constructor cannot be created, but virtual destructor can be created.
5) A base pointer can points to any type of derived objects but derived pointer cannot points to the base class.
6) If a virtual function is defined in the base class, then it is not needed to 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 in the base class.
- the notation =0 indicate virtual function is a pure virtual function.

syntax:

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

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<<"\n derived show:";
           }
};

Void main()
{
    Base *bptr;
    derived d;
    clrscr();

     bptr=&d;

     bptr -> show();

     getch();
}
```

PAGE No.
DATE

o|p
derived shown:


VJTech Academy
VJ
Inspiring Your Success...