

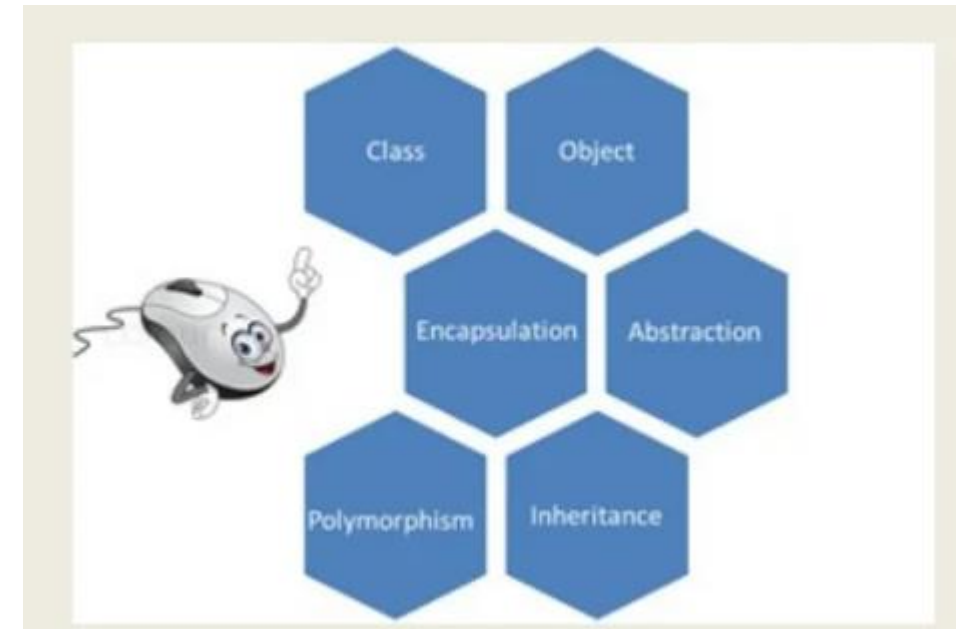


VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)
CHENNAI

BCSE102L –Structured and Object oriented Programming (Theory)

OBJECT ORIENTED CONCEPTS - FEATURES

MODULE V



Why Do We Need Object-Oriented Programming?

- Object-Oriented Programming was developed because limitations were discovered in earlier approaches to programming.
- To appreciate what OOP does, we need to understand what these limitations are and how they arose from traditional programming languages.



Procedure-Oriented Programming

- C, Pascal, FORTRAN, and similar languages are procedural languages.
- Each statement in the language tells the computer to do something:
 - get some input
 - add these numbers
 - divide by 6
 - display that output
- A program in a procedural language is a list of **instructions**.

Procedure-Oriented Programming

Division into Functions:

- Procedural program is divided into **functions**
- Each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.
- The idea of breaking a program into functions can be further extended by grouping a number of functions together into a larger entity called a **module**.

Procedure-Oriented Programming

Division into Functions:

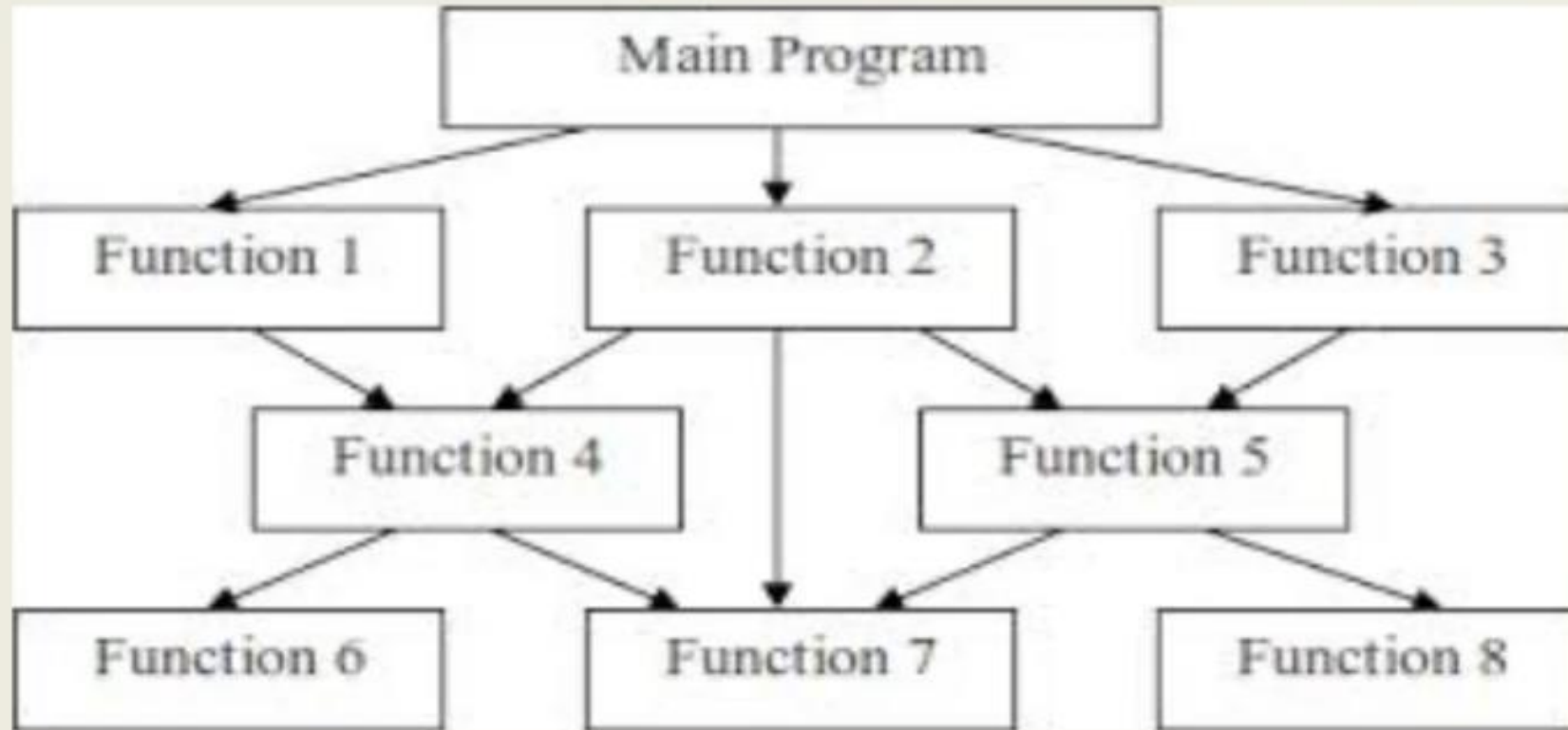


Fig 1: Structure of Procedure-Oriented Programming

Procedure-Oriented Programming

- In Multi-function program important data items are placed as **global** so that they may be accessed by all functions.
- Each function may have its own **local** data.

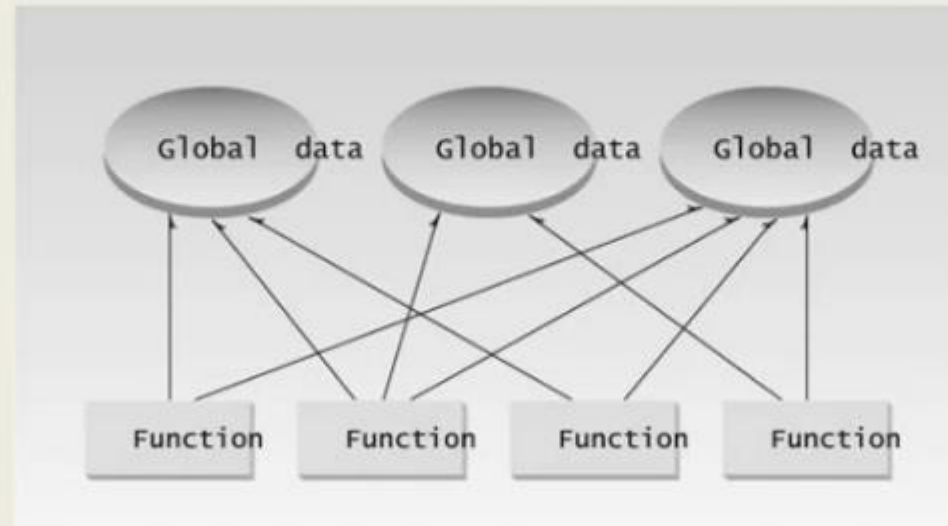


Fig 2: Procedural paradigm

Procedure-Oriented Programming

Drawbacks:

- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function.
- We can access the data of one function from other since, there is no protection.
- In large program it is very difficult to identify what data is used by which function.
- Similarly, if new data is to be added, all the function needed to be modified to access the data.
- Does not model real world problem very well.

Procedure-Oriented Programming

Characteristics:

- Emphasis is on doing things (**algorithms**).
- Large programs are divided into smaller programs known as **functions**.
- Most of the functions share **global** data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs **top-down approach** in program design.

Top-Down Approach

- **Top-down decomposition** is the process of breaking the overall procedure or task into component parts (**modules**) and then subdivide each component module until the lowest level of detail has been reached.
- **Example** :The payroll system of a company can contain the following modules or tasks
 - Master file
 - Earnings
 - Deductions
 - Taxing
 - Net earning
 - Print reports

Object-Oriented Programming

- OOP was introduced to overcome flaws in the procedural approach to programming.
- Such as lack of **reusability** & **maintainability**.
- Fundamental idea behind object-oriented languages is to combine into a single unit both **data** and the **functions that operate on that data**.
- Such a unit is called an **object**.

Object-Oriented Programming

- In OOP, problem is divided into the number of entities called **objects** and then builds data and functions around these objects.
- It ties the data more closely to the functions that operate on it, and protects it from accidental modification from the outside functions.
- Data of an object can be accessed only by the functions associated with that object.
- Communication of the **objects** done through **function**.

Object-Oriented Programming

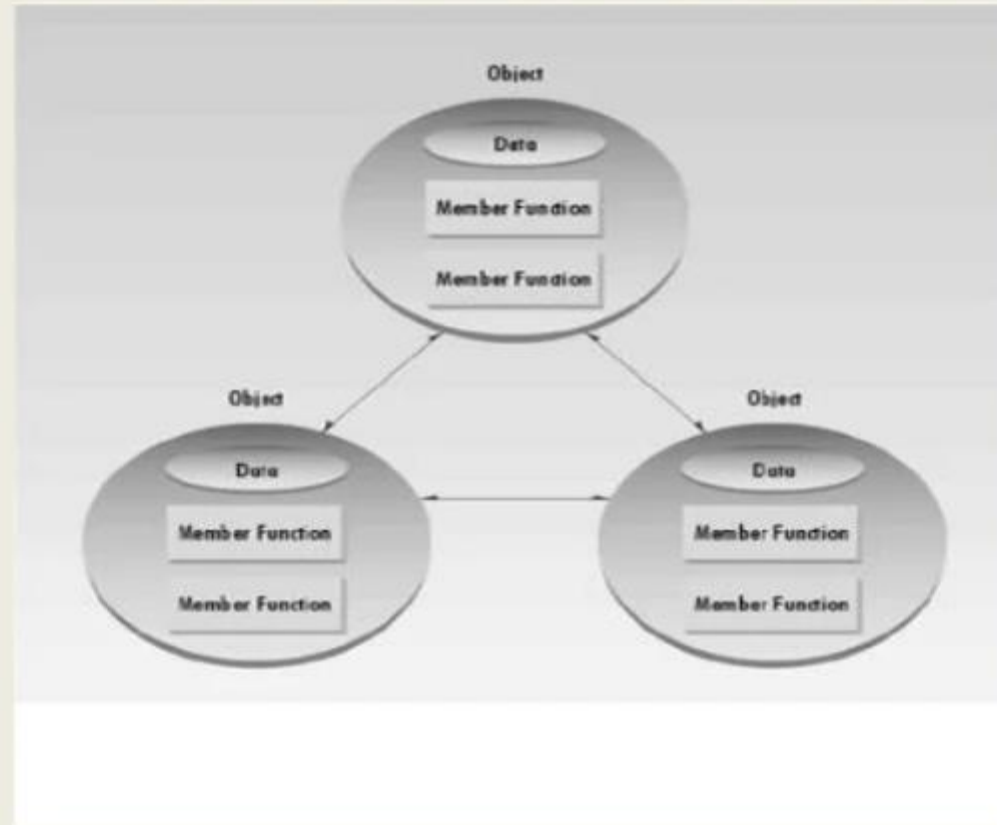


Fig 3: Object-Oriented paradigm

Object-Oriented Programming

Characteristics:

- Emphasis on data rather than procedure.
- Programs are divided into entities known as **objects**.
- Data Structures are designed such that they characterize objects.
- Functions that operate on data of an object are tied together in data structures.
- Data is hidden and cannot be accessed by external functions.
- Objects communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows **bottom up design** in program design.

Bottom up approach

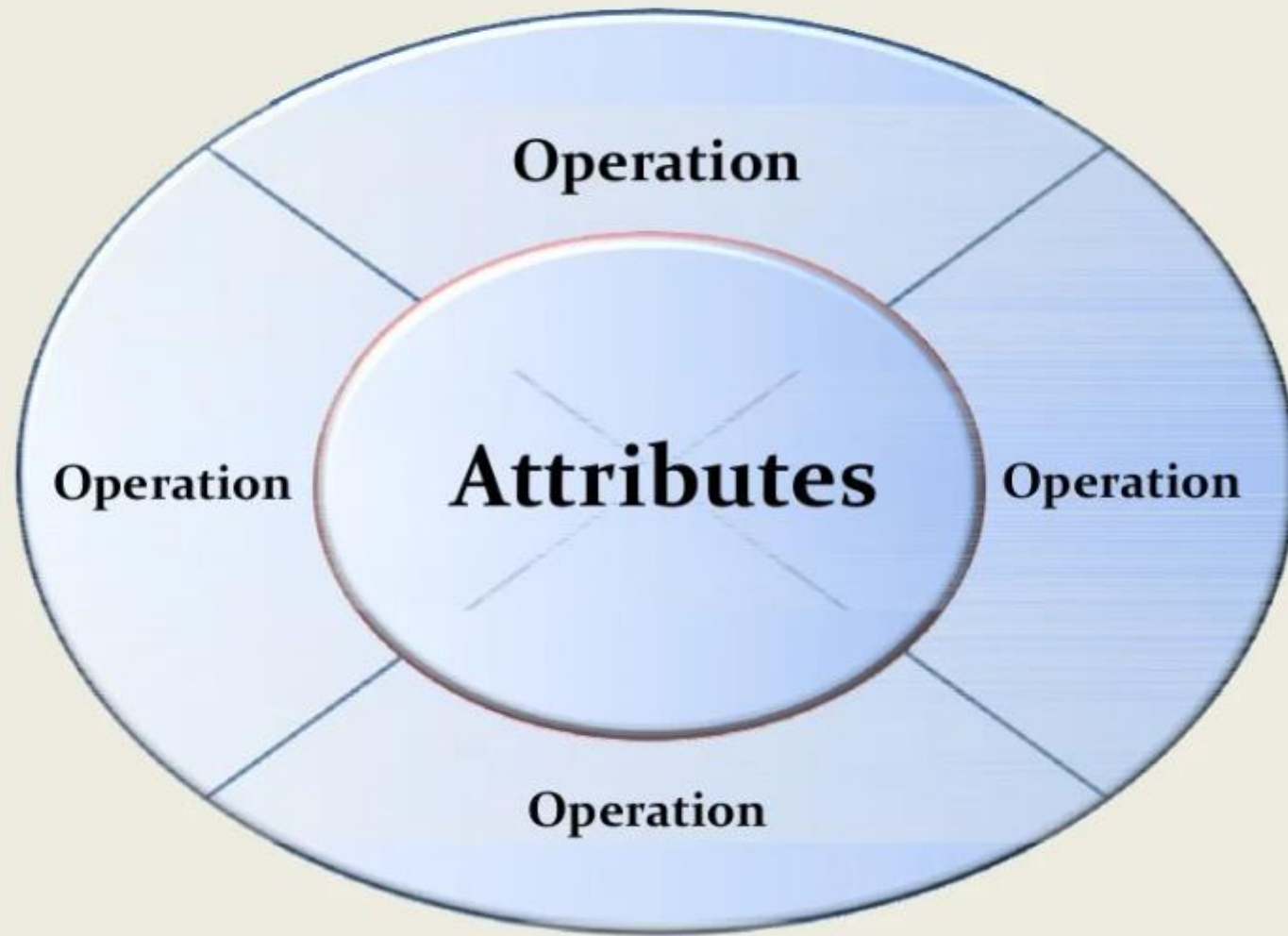
- Reverse top-down approach.
- Lower level tasks are first carried out and are then integrated to provide the solution of a single program.
- Lower level structures of the program are evolved first then higher level structures are created.
- It promotes code reuse.
- It may allow unit testing.

Basic Concepts of oops

1. Objects
2. Classes
3. Data Abstraction
4. Data Encapsulation
5. Inheritance
6. Polymorphism
7. Dynamic binding
8. Message Passing



Objects



Objects

- Objects are the basic run-time entities of an object oriented system.
- They may represent a person, a place or any item that the program must handle.
- **Example:**

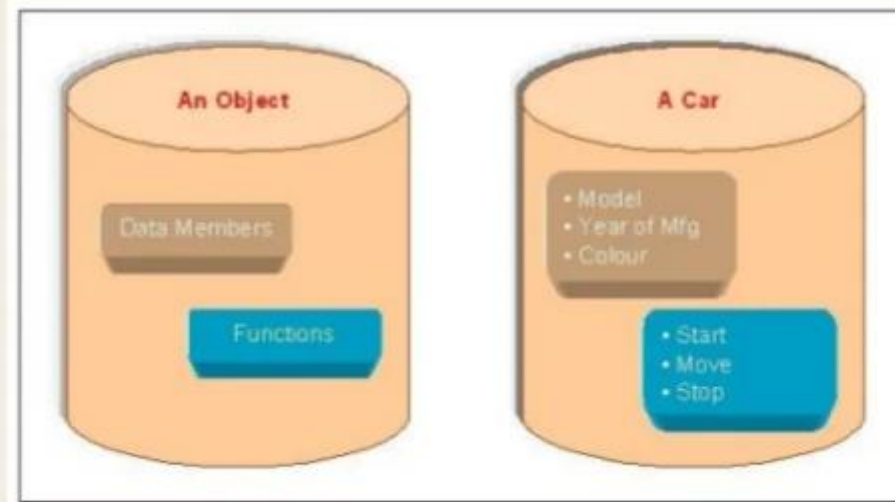
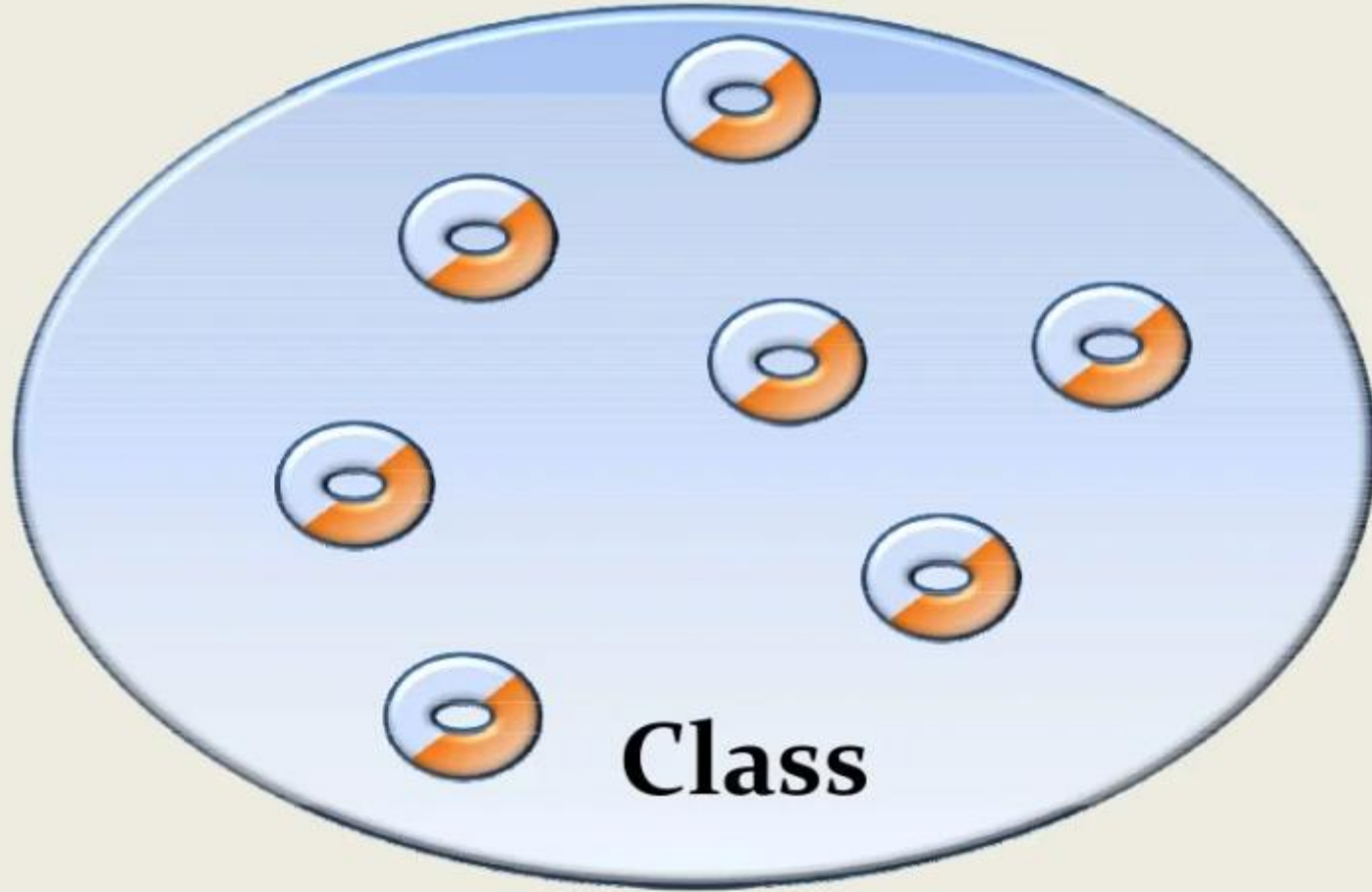


Fig 4: Representation of object.

Objects

- When a program is executed, the objects interact by sending messages to one another.
- For e.g. if “customer” and “account” are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance.
- Each object contains data, and code to manipulate the data.

Classes



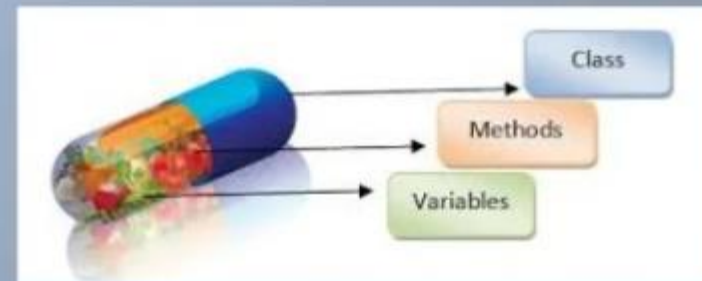
Classes

- Classes are **user-defined** data types and it behaves like built in types of programming language.
- Object contains code and data which can be made user define type using class.
- Objects are **variables** of class.
- Once a class has been defined we can create any number of objects for that class.
- A class is collections of **objects** of similar type.

Classes

- We can create object of class using following syntax,
- Syntax: `class-name object-name;`
- Here class name is class which is already created. Object name is any user define name. For example, if Student is class,
- Example: `Student ram, sham;`
- In example `ram` and `sham` are name of objects for class `Student`. We can create any number of objects for class.

Encapsulation



Encapsulation



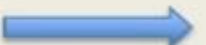
- Encapsulation is the first pillar or principle of object-oriented programming.
- In simple words, “**Encapsulation** is a process of binding **data members** (variables, properties) and **member functions** (methods) into a single unit”.
- And **Class** is the best example of encapsulation.

Encapsulation

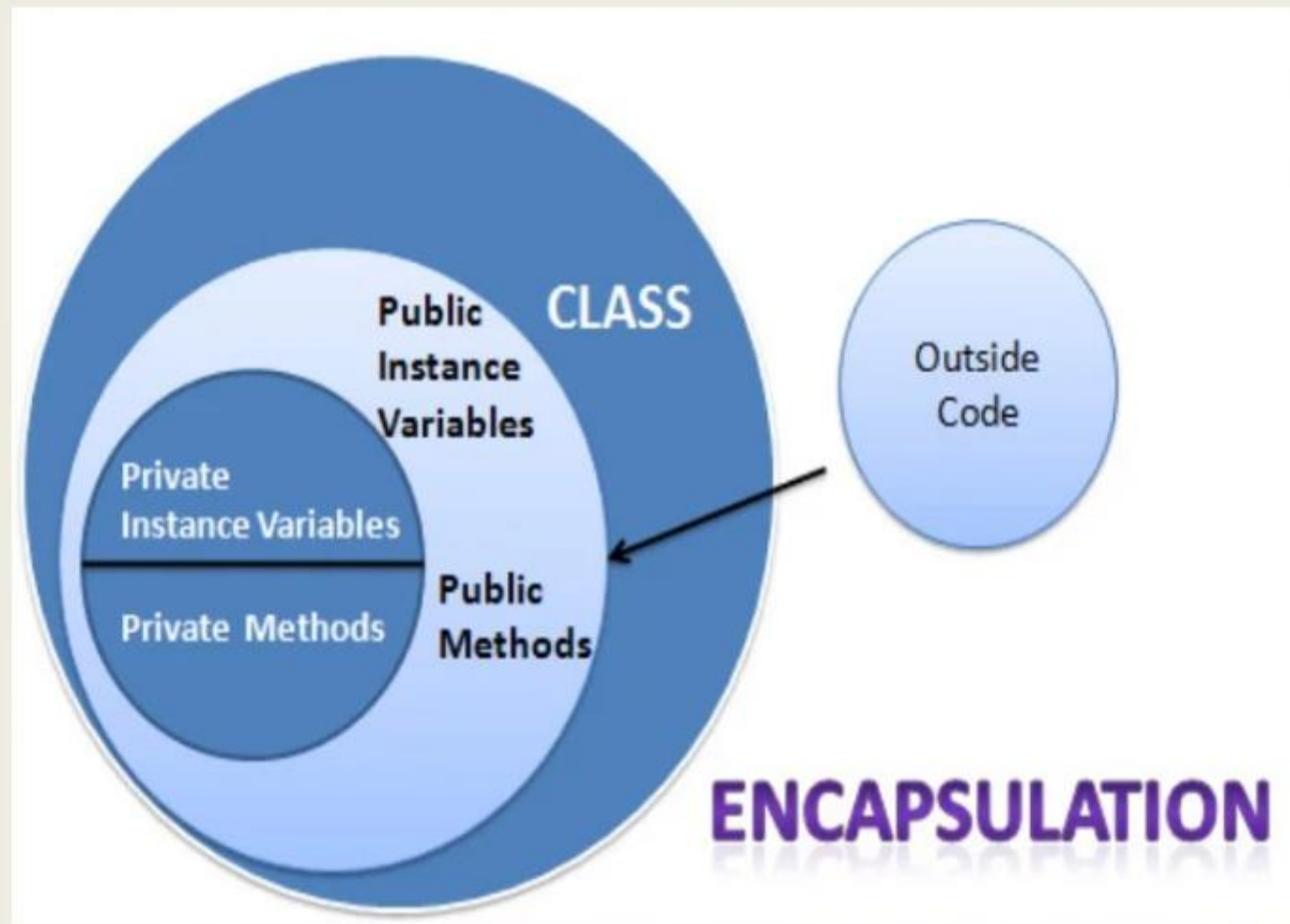
- For example: **Medical store**
- Lets say you have to buy some **medicines**. You go to the **medical store** and ask the **chemist** for the medicines.
- Only the **chemist** has access to the **medicines** in the store based on your prescription.
- The **chemist** knows what **medicines** to give to you.
- This reduces the risk of you taking any medicine that is not intended for you.

Encapsulation

In this example,

- MEDICINES  Member Variables.
- CHEMIST  Member Methods.
- You  External Application or piece of Code.

Encapsulation



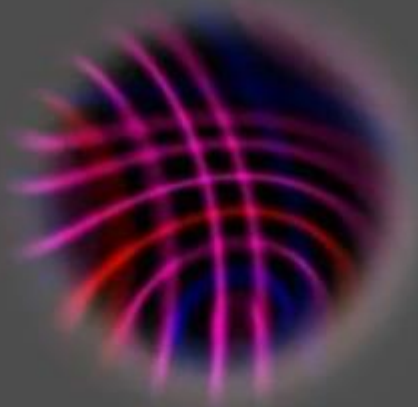
Encapsulation

- Through Encapsulation, Data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- Encapsulation solves the problem at the implementation level.
- A class can specify how accessible each of its members (variables, properties, and methods) is to code outside of the class.

Encapsulation

- So **encapsulation** means hiding the important features of a class which is not been needed to be exposed outside of a class and exposing only necessary things of a class.
- Here hidden part of a class acts like **Encapsulation** and exposed part of a class acts like **Abstraction**.

Abstraction



Abstraction

- **Abstraction** refers representation of necessary features without including more details or explanations.
- **Data abstraction** is a programming (and design) technique that relies on the separation of **interface** and **implementation**.

Abstraction

- When you **press a key on your keyboard the character appears on the screen**, you need to know only this, but How exactly it works based on the electronically is not needed.
- This is called **Abstraction**.

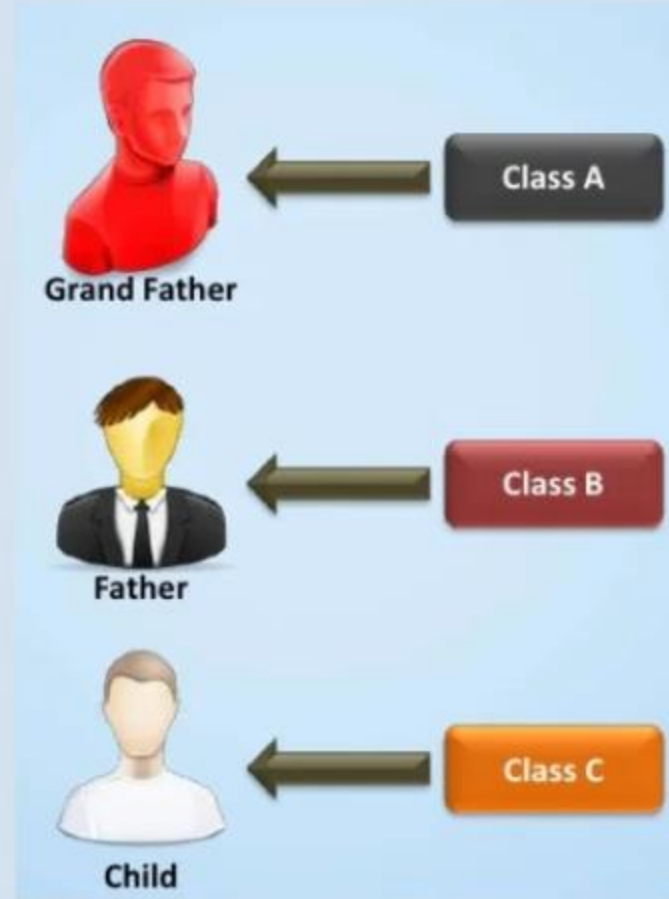
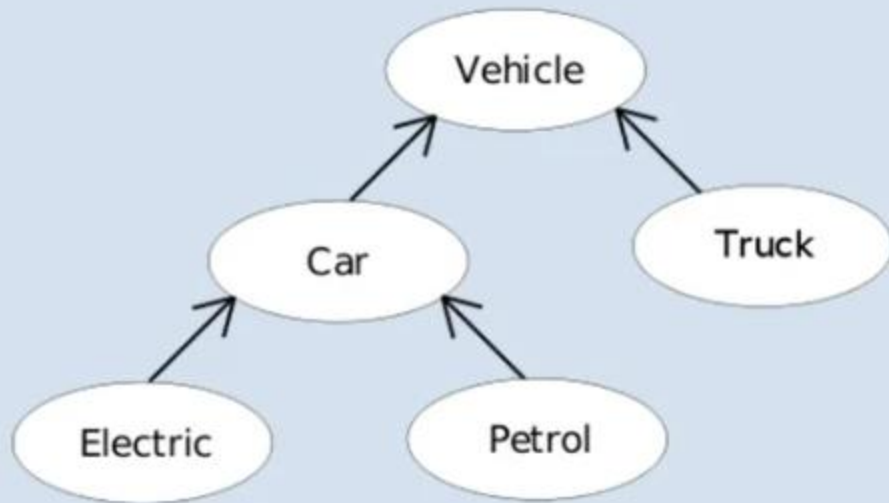


Abstraction

- Another **Example** is when you use the **remote control of your TV**, you do not bother about how pressing a key in the remote changes the channel on the TV. You just know that pressing the + volume button will increase the volume!



Inheritance



Inheritance

- The mechanism of deriving a **new** class from an **old** class is called **inheritance** or **derivation**.
- The **old** class is known as **base** class while **new** class is known as **derived** class or sub class.
- The inheritance is the most **powerful** features of OOP.

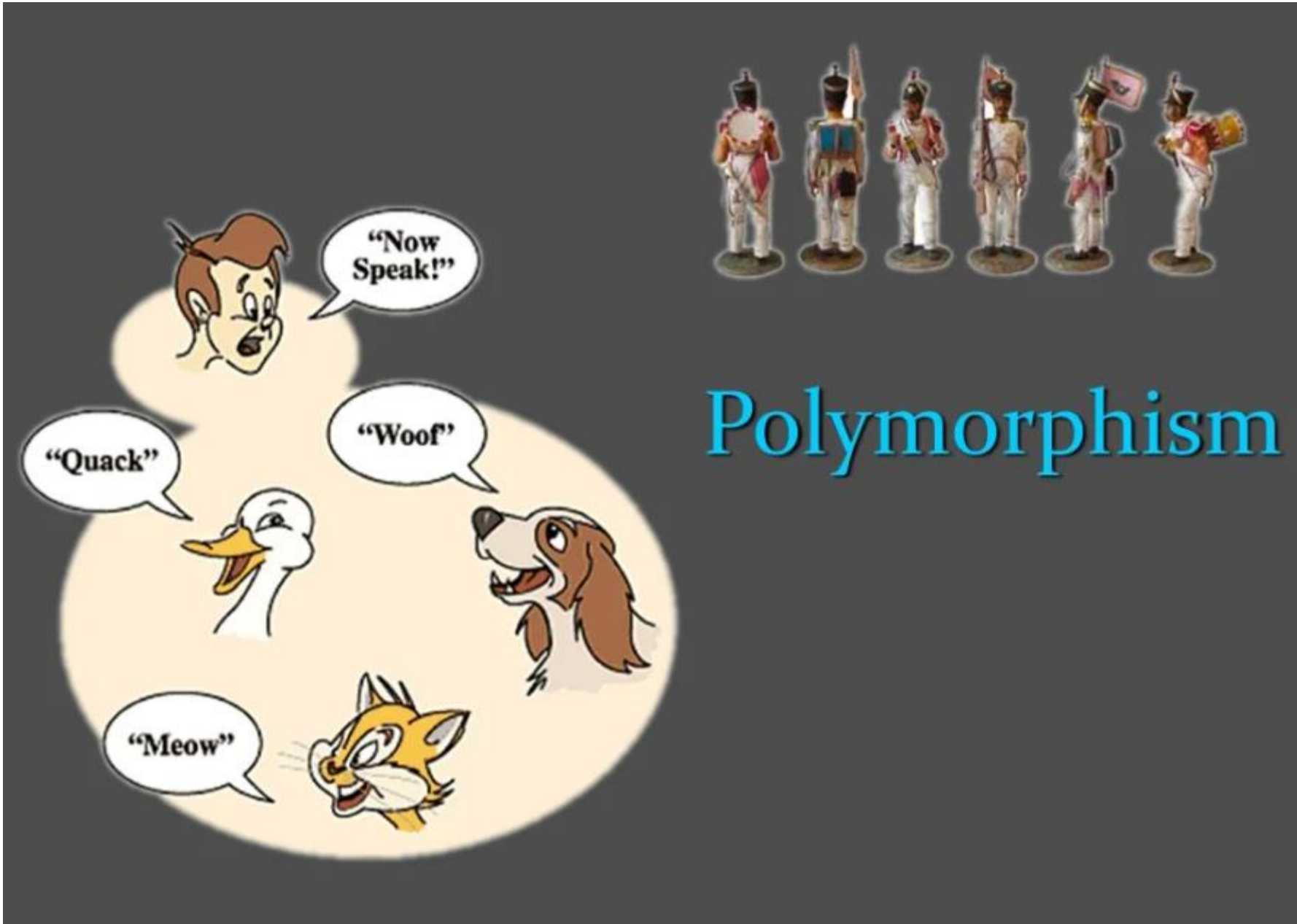
Inheritance

- For Example:
- Consider an example of **family** of three members having a mother, father & son named Jack.
- Jack **father** : **tall** and dark
- Jack **Mother** : Short and **fair**
- Jack is **tall** and **fair**, so he is said to have inherited the features of his **father** and **mother** resp.

Inheritance

- Through effective use of inheritance, you can **save lot of time** in your programming and also **reduce errors**
- Which in turn will increase the **quality** of work and **productivity**.

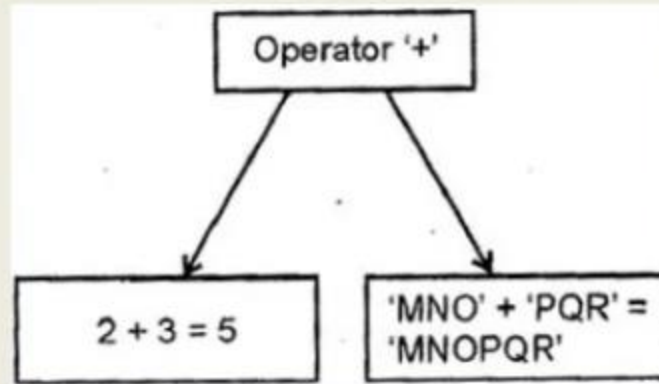




Polymorphism

Polymorphism

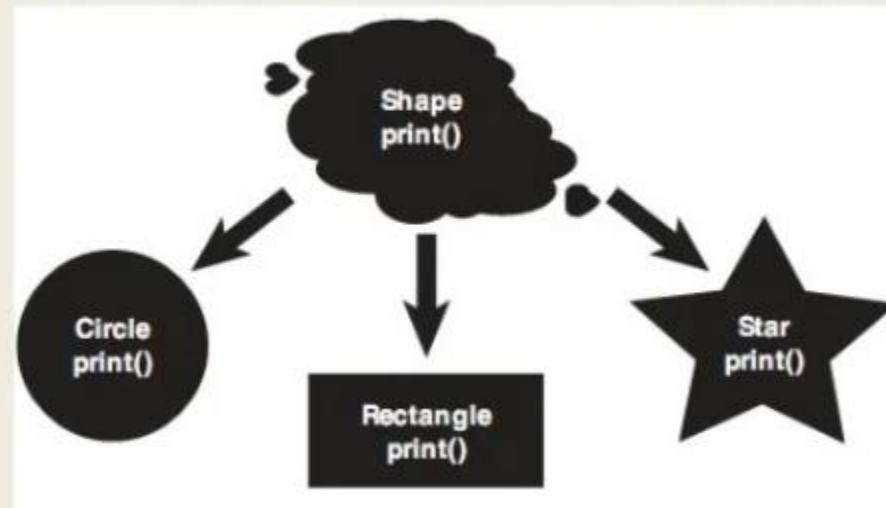
- **Polymorphism** is a **Greek** term which means ability to take more than one form.
- For example, **+** is used to make sum of two **numbers** as well as it is used to combine two **strings**.



- This is known as **operator overloading** because same operator may behave differently on different **instances**.

Polymorphism

- Same way **functions** can be overloaded.
- For example, `sum ()` function may takes two arguments or three arguments etc. i.e. **sum** (5, 7) or **sum** (4, 6, 8).
- Single function **print()** draws different objects.



Dynamic binding

- Binding means **link** between **procedure** call and **code** to be execute.
- It is the **process** of linking of a function call to the actual code of the function at **run-time**.
- That is, in dynamic binding, the actual code to be executed is not known to the compiler until run-time.
- It is also known **late binding**.

Dynamic binding

- Binding means **link** between **procedure** call and **code** to be execute.
- It is the **process** of linking of a function call to the actual code of the function at **run-time**.
- That is, in dynamic binding, the actual code to be executed is not known to the compiler until run-time.
- It is also known **late binding**.

Dynamic binding

- For example, compiler comes to know at runtime that which function of **sum** will be call either with **two arguments** or with **three arguments**.

Message Passing

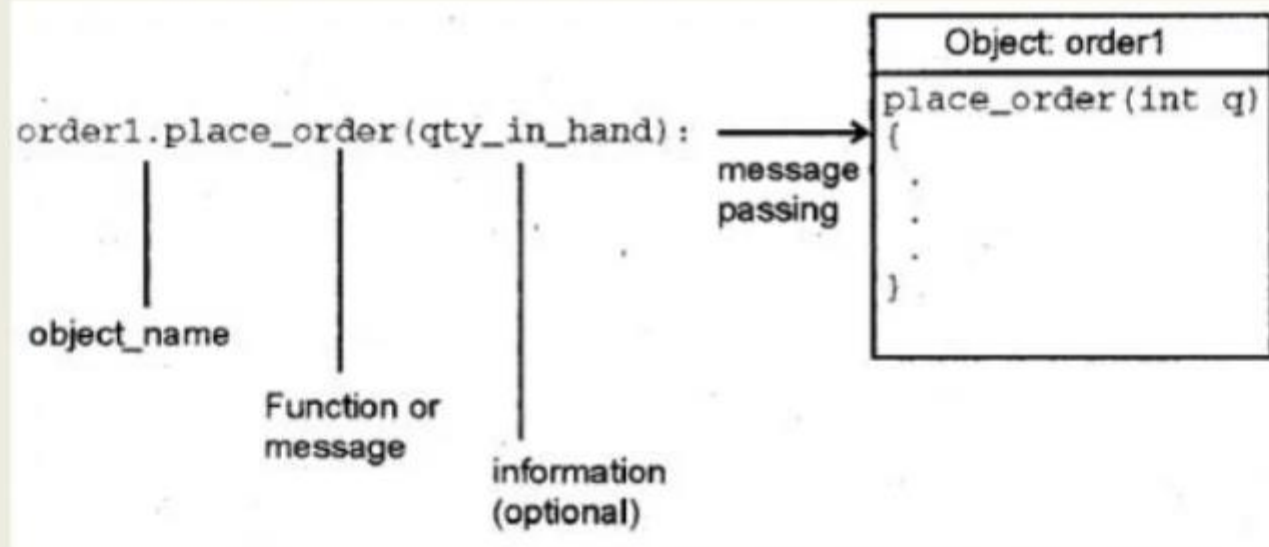


Message Passing

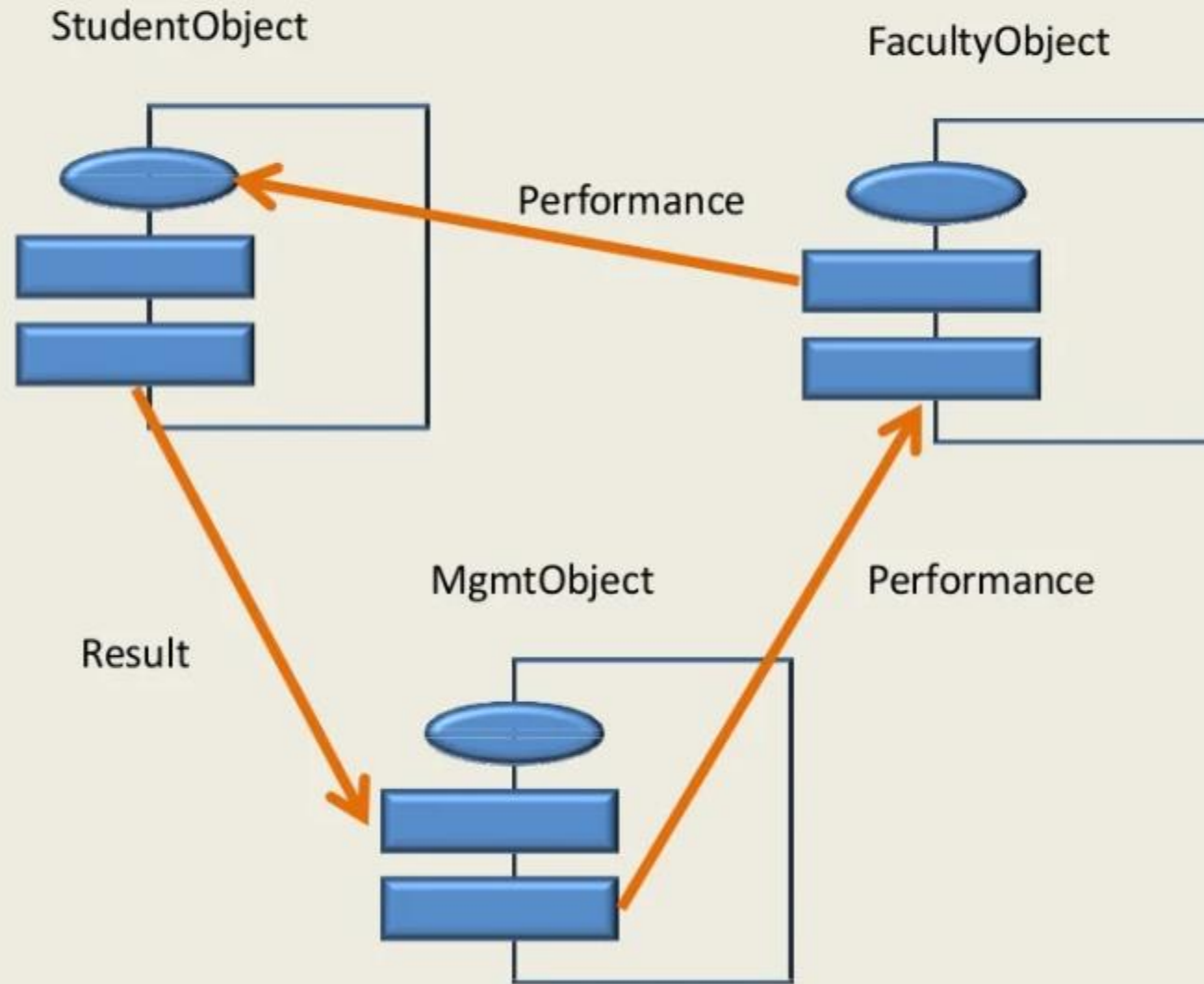
- Objects can **communicate** with each others by passing message same as people passing message with each other.
- Objects can send or receive message or information.
- Message passing involves the following basic steps:
 - Creating classes that define objects & their behavior.
 - Creating objects from class definitions.
 - Establishing communication among objects.
- Concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

Message Passing

- For example, consider two **classes** *Product* and *Order*. The object of the *Product* class can **communicate** with the object of the *Order* class by sending a request for placing order.



Message Passing



Benefits of OOP

- User can create new data type or users define data type by making class.
- Code can be reuse by using inheritance.
- Data can be hiding from outside world by using encapsulation.
- Operators or functions can be overloaded by using polymorphism, so same functions or operators can be used for multitasking.

Benefits of OOP

- Object oriented system can be easily upgrade from small system to large system.
- It is easy to partition work for same project.
- Message passing techniques make communication easier.
- Software complexity can be easily managed.
- Maintenances cost is less.
- Simple to implement.

Areas for applications of OOP

- Real time systems
- Simulation and modeling
- Object oriented database
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural network and parallel programming
- Decision support system
- Office automation system
- CIM / CAM / CAD systems



CLASS & OBJECT

- Class: A Class is a user defined data type to implement an abstract object. Abstract means to hide the details. A Class is a combination of data and functions.
- Data is called as data members and functions are called as member functions.

■ Abstract data type:-

- A data type that separates the logical properties from the implementation details called Abstract Data Type(ADT).
- An abstract data type is a set of object and an associated set of operations on those objects.
- ADT supports data abstraction, encapsulation and data hiding.

- Examples of ADT are:-

- Boolean

- Integer

- Array

- Stack

- Queue

- Tree search structure



built in ADT



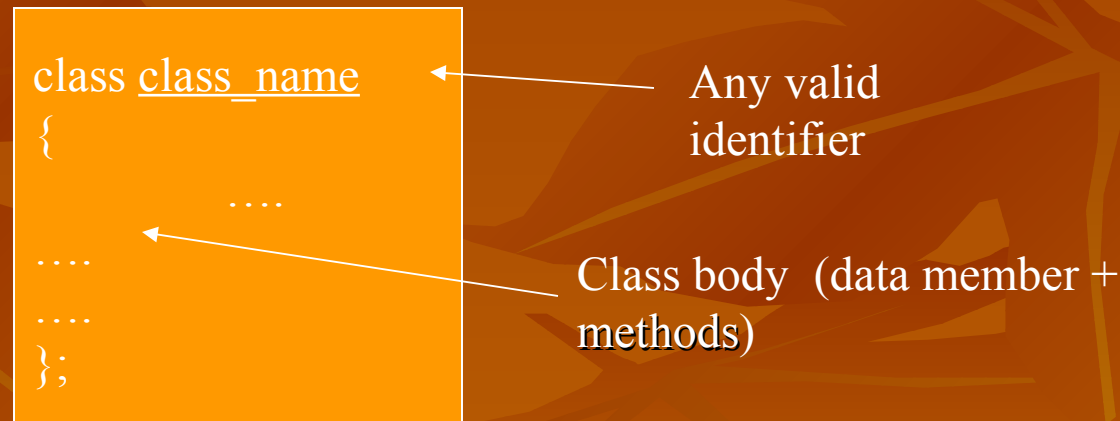
User defined
ADT

- Boolean {operations are AND,OR,NOT and values are true and false}

- Queues {operations are create , dequeue,inqueue and values are queue elements}

Class definition

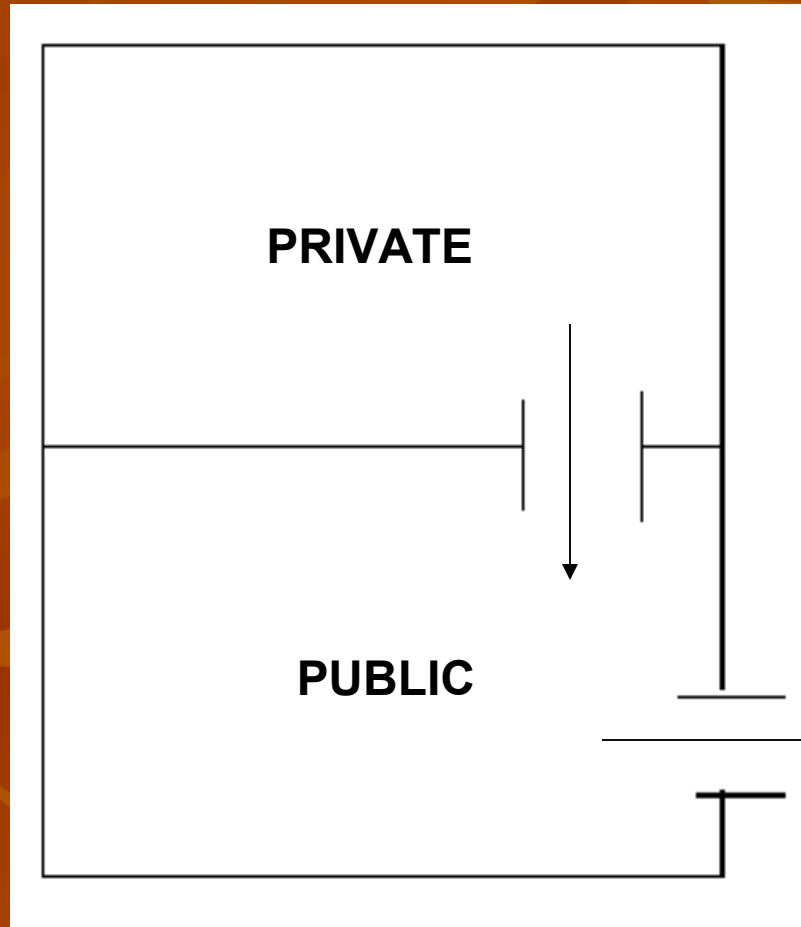
- A class definition begins with the keyword *class*.
- The body of the class is contained within a set of braces, { } ; (notice the semi-colon).



- Within the body, the keywords *private:* and *public:* specify the access level of the members of the class.
 - the default is *private*.
- Usually, the data members of a class are declared in the *private:* section of the class and the member functions are in *public:* section.

- Data member or member functions may be public, private or protected.
- Public means data members or member functions defining inside the class can be used at outside the class.(in different class and in main function)
- Member access specifiers
 - public:
 - can be accessed outside the class directly.
 - The public stuff is *the interface*.

- **private:**
 - Accessible only to member functions of class
 - Private members and methods are for internal use only.
- Private means data members and member functions can't be used outside the class.
- Protected means data member and member functions can be used in the same class and its derived class (at one level) (not in main function).



```
class class_name
```

```
{
```

```
private:
```

```
...
```

```
...
```

```
...
```

```
public:
```

```
...
```

```
...
```

```
...
```

```
};
```

private members or methods

Public members or methods

- This class example shows how we can encapsulate (gather) a circle information into one package (unit or class)

```
class Circle
{
    private:
        double radius;
    public:
        void setRadius(double r);
double getDiameter();
        double getArea();
        double getCircumference();
};
```

No need for others classes to access and retrieve its value directly. The class methods are responsible for that only.

They are accessible from outside the class, and they can access the member (radius)

Class Example (Problem)

```
#include<iostream.h>
class student
{
int rollno;
char name[20];
};
```

```
void main()
{
student s;
cout<<"enter the rollno.:";
cin>>s.rollno;
cout<<"enter the name:";
gets(s.name);
cout<<"rollno:"<<s.rollno;
cout<<"\nname:";
puts(s.name);
}
```

Class Example (Solution)

```
#include<iostream.h>
class student
{
public:
int rollno;
char name[20];
};
```

```
void main()
{
student s;
cout<<"enter the rollno.:";
cin>>s.rollno;
cout<<"enter the name:";
gets(s.name);
cout<<"rollno:"<<s.rollno;
cout<<"\nname:";
puts(s.name);
}
```

Implementing class methods

- There are two ways:
 1. Member functions defined outside class
 - Using Binary scope resolution operator (: :)
 - “Ties” member name to class name
 - Uniquely identify functions of particular class
 - Different classes can have member functions with same name
 - Format for defining member functions

```
ReturnType ClassName : : MemberFunctionName ( ) {  
    ...  
}
```

Member Function Defining Inside the Class

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

```
class student
```

```
{  
int rollno;  
char name[20];
```

Data Members (Private : in this example)

```
public:
```

```
void getdata()
```

```
{  
cout<<"enter the rollno.:";  
cin>>rollno;  
cout<<"enter the name:";  
gets(name);
```

Member Functions (Public: in this example)

```
}
```

```
void putdata()
```

```
{  
cout<<"rollno:"<<rollno;  
cout<<"\nname:";  
puts(name);
```

```
}
```

```
};
```

```
void main()
```

```
{  
student s;  
s.getdata();  
s.putdata();
```

Calling member function

```
}
```

Member Function Defining Outside the Class

```
#include<iostream.h>
class student
{
int rollno;
char name[20];
public:
void getdata();
void putdata();
};
void student :: getdata()
{
cout<<"enter the rollno.:";
cin>>rollno;
cout<<"enter the name:";
gets(name);
}
```

```
void student :: putdata()
{
cout<<"rollno:"<<rollno;
cout<<"\nname:";
puts(name);
}
void main()
{
student s;
s.getdata();
s.putdata();
}
```


Characteristics of member function

- Different classes have same function name. the “membership label” will resolve their scope.
- Member functions can access the private data of the class .a non member function cannot do this.(friend function can do this.)
- A member function can call another member function directly, without using the dot operator.

Accessing Class Members

- Operators to access class members
 - Identical to those for **structs**
 - Dot member selection operator (.)
 - Object
 - Reference to object
 - Arrow member selection operator (->)
 - Pointers

Static members

- The data and functions of the class may be declared static in the class declaration.
- The static data members have similar properties to the C static variable.
- The static data members is initialized with zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static member function

- Like static data members we can also declare static member functions.
- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the class (instead of its objects) as follows
- `Class name:: function name.`

Example of static members

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

```
Class test
```

```
{  
    Int code;  
    Static int count;  
    Public:  
    Void setcode()  
    {  
        Code=++count;  
    }  
    Void showcode()  
    {  
        Cout<<"object number "<<code<<endl;  
    }  
    Static void showcount()  
    {  
        Cout<<"count :"<<count;  
    }  
};
```

```
Int test::count;
```

```
Int main()
```

```
{  
    test t1,t2;  
    t1.setcode();  
    t2.setcode();  
    test::showcount();  
    test t3;  
    t3.setcode();  
    test::showcount();  
    t1.showcode();  
    t2.showcode();  
    t3.showcode();  
    Return 0;  
}
```


Class inside a function

- When a class declared within a function, it is known as local class.
- A local class is known only to that function and unknown outside it.
- All member functions must be defined within the class declaration.
- The local class may not use local variables of the function in which it is declared except static and extern local variables declared within the function.
- No static variables may be declared inside a local class.
- Due to these restrictions local class is not popular in C++ programming.

Objects

- An object is an instance of a class.
- An object is a class variable.
- It can be uniquely identified by its name.
- Every object has a state which is represented by the values of its attributes. These states are changed by functions which are applied to the object.

State identity and behavior of objects

- Every object have *identity* , *behaviour* and *state*.
- The identity of object is defined by its name, every object is unique and can be differentiated from other objects.
- The behavior of an object is represented by the functions which are defined in the object's class. These function show the set of action for every objects.
- The state of objects are referred by the data stored within the object at any time moment.

Creating an object of a Class

- Declaring a variable of a class type creates an **object**. You can have many variables of the same type (class).
 - *Also known as Instantiation*
 - Once an object of a certain class is instantiated, a new memory location is created for it to store its data members and code
 - You can instantiate many objects from a class type.
 - Ex) Circle c; Circle *c;
- Class item
- ```
{
```

```
.....
,,,,,,,,,,,,,
} x,y,z;
```

We have to declared objects close to the place where they are needed because it makes easier to identify the objects.

# Object types

- There are four types of objects

1. External (global) objects

1. This object have the existence throughout the lifetime of the program and having file –scope.

2. Automatic(local)objects

1. Persistent and visible only throughout the local scope in which they are created.

3. Static objects

1. Persistent throughout a program but only visible within their local scope.

4. Dynamic objects

1. Lifetime may be controlled within a particular scope.



# Memory Allocation of Object

```
class student
{
int rollno;
char name[20];
int marks;
};
student s;
```



24 bytes      s

# Array of objects

- The array of class type variable is known as array of object.
- We can declare array of object as following way:-  
Class \_name object [length];  
Employee manager[3];
  1. We can use this array when calling a member function
  2. Manager[i].put data();
  3. The array of object is stored in memory as a multi-dimensional array.

# Object as function arguments

- This can be done in two ways:-
  - A copy of entire object is passed to the function.
    - ( pass by value)
  - Only the address of the object is transferred to the function. (pass by reference)

## ( pass by value)

- A copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call function.

## (pass by reference)

- When an address of object is passed, the called function works directly on the actual object used in the call. Means that any change made inside the function will reflect in the actual object.

# Passing Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
void sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
void Complex :: sum (complex A, complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```



# Passing Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
void sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
void Complex :: sum (Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}
```

```
void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```



X



Y



Z

# Passing Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
void sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
void Complex :: sum (Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}
```

```
void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```

|   |
|---|
| 5 |
| 6 |

X

|   |
|---|
| 7 |
| 8 |

Y

|  |
|--|
|  |
|  |

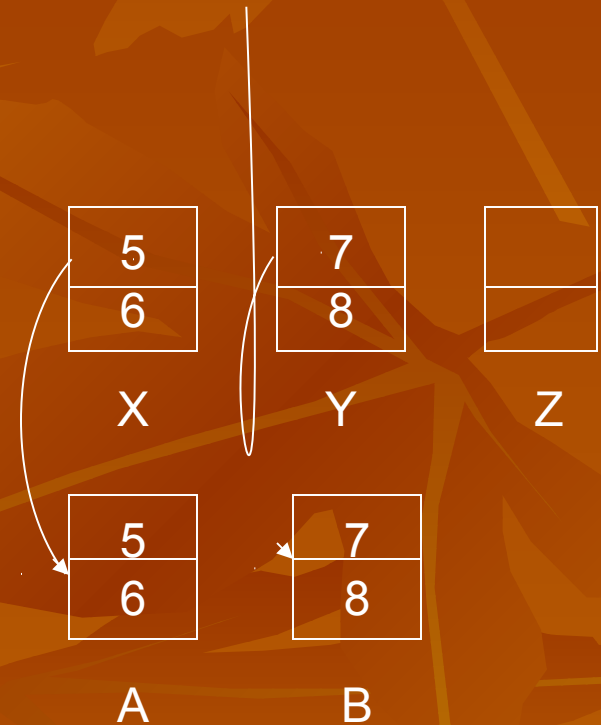
Z

# Passing Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
void sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
void Complex :: sum (Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}
```

```
void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```

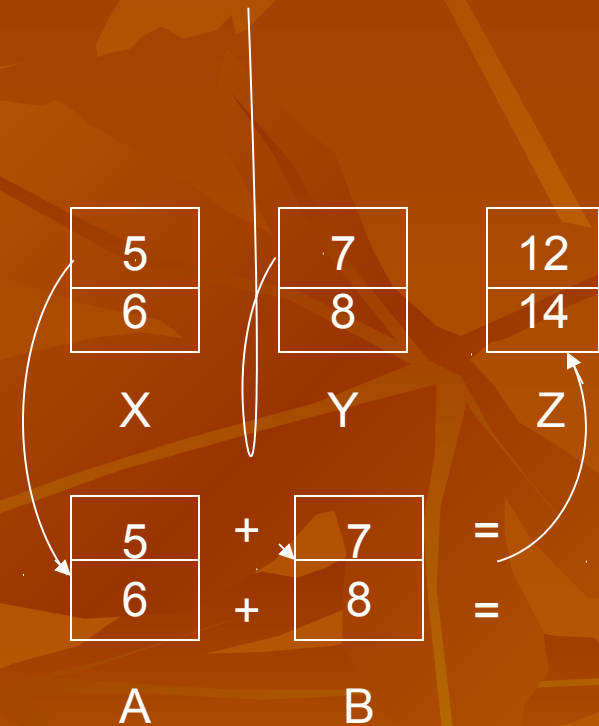


# Passing Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
void sum(Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
void Complex :: sum (Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}
```

```
void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```



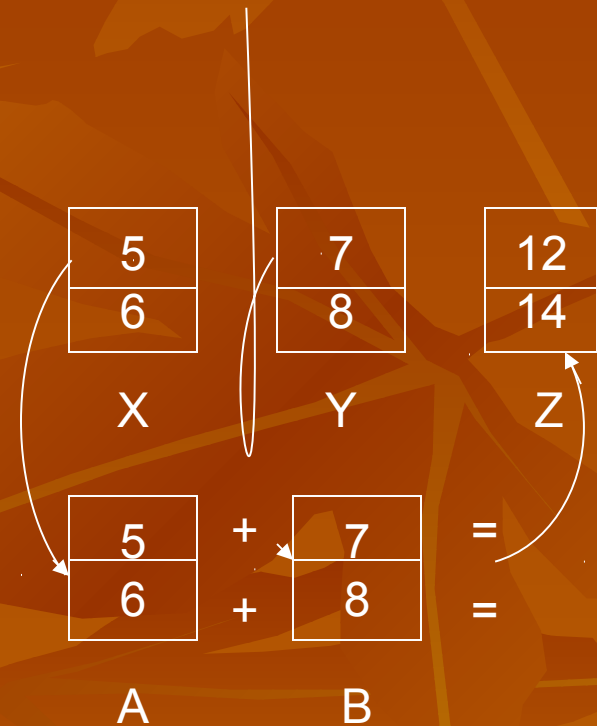
# Passing Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
void sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
void complex :: sum (Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}
```

```
void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```

12 + 14 i





# Returning Object

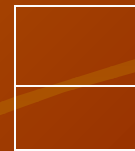
```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```

# Returning Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

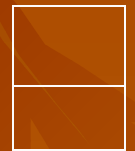
```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```



X



Y



Z

# Returning Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```

|   |
|---|
| 5 |
| 6 |

X

|   |
|---|
| 7 |
| 8 |

Y

|  |
|--|
|  |
|  |

Z

# Returning Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

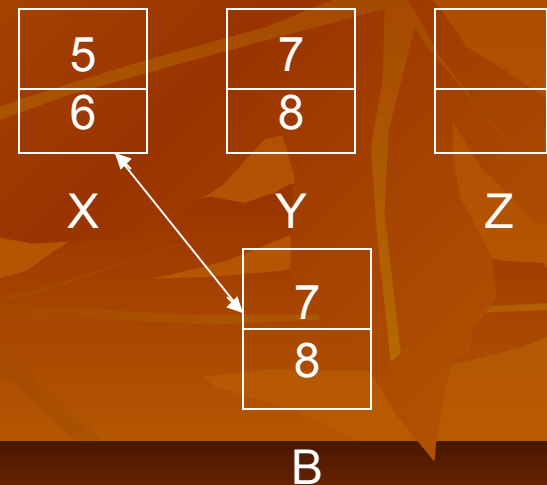
```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```



# Returning Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```

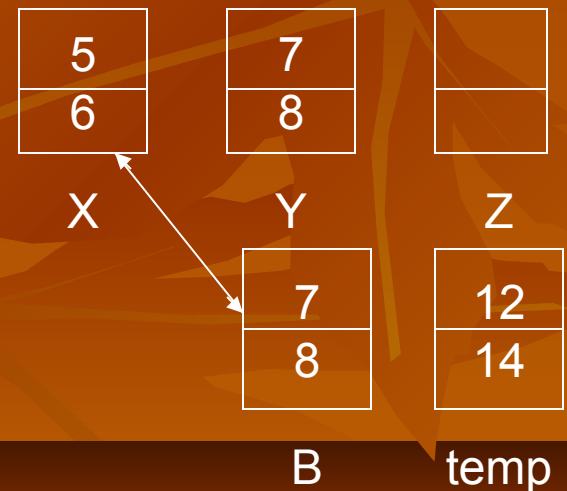




# Returning Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

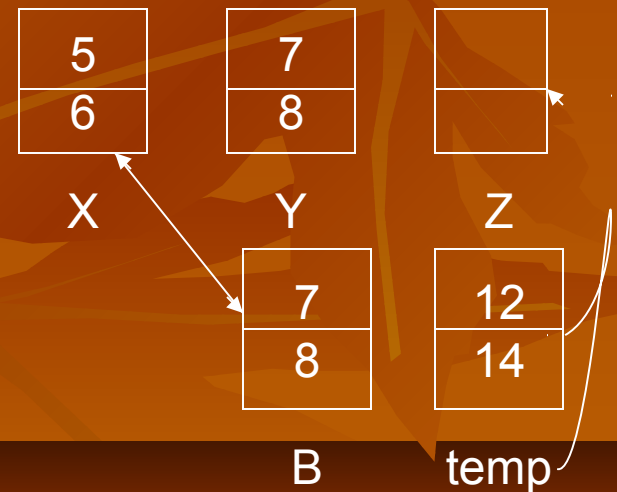
```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```



# Returning Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

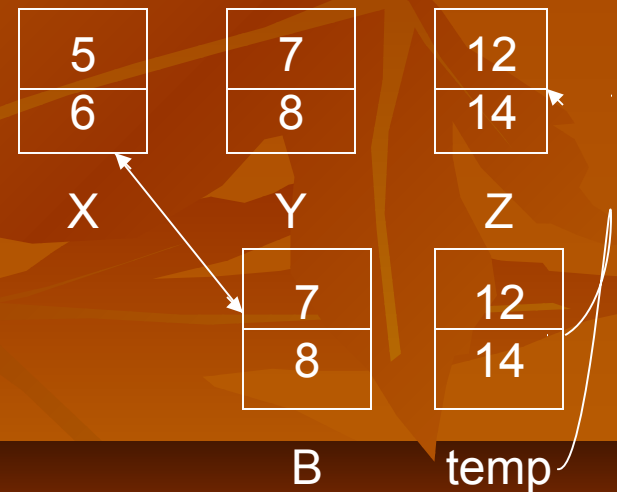
```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```



# Returning Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
}
```

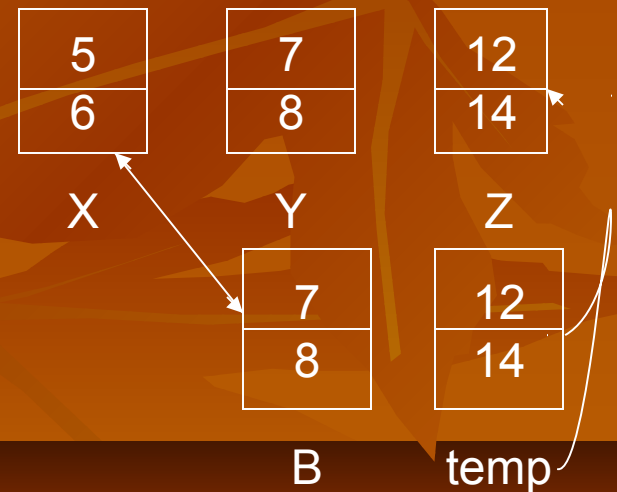
```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```



# Returning Object

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
Complex sum (Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex Complex :: sum (Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```



**12 + 14 i**

# C++ garbage collection

- In c++ the garbage collection task is accomplished by mark and sweep algorithm.
- In this approach the garbage collector periodically examines every single pointer in our program and find that the memory is still in use. At the end of the cycle, any memory that has not been marked is deemed to be not in use and is freed.



# Steps to implement the garbage collection

- Mark and sweep algorithm could be implemented in c++ if we are willing to do the following:
  1. Register all pointers with the garbage collector so that it can easily walk through the list of all pointers.
  2. Sub-class all objects from a mix-in class, that allows the garbage collectors to mark an object as in-use.
  3. Protect concurrent access to objects by making sure that no changes to pointers can occur while the garbage collector is running.

# Memory management in c++

- Ways of memory allocation in c++
- Static memory allocation
- The memory allocation for variables ,during compilation time itself is known as static memory allocation.
- Once the memory allocated at the compile time then it can not be expanded nor be compressed to accommodate more or less data during program execution.
- The size of memory to be allocated is known before compile time and is fixed it can not be altered during execution.
- `Int a[10];`

# Dynamic memory allocation

- The dynamic memory allocation is carried-out in c++ using two operators “new” and “delete”.these operators are use to allocate and free memory at run time.
- Dynamic memory allocation helps in memory saving and easy to change memory allocation.
- In c++ dynamic memory allocation is control by NEW and DELETE operator.
- The new operator return the memory pointer to the pointer variable.

- **Syntax:**

  - Ptr= new data type;

  - Delete Ptr;

  - Ptr is pointer and data type is valid data type

- The difference between NEW and malloc function is that NEW automatically calculates the size of operand , dos not use size of operator and NEW does not require an explicit type cast.

- **Versions of NEW and DELETE**

  - in c++ NEW and DELTE should be used like malloc and free to ensure the proper calling of constuctor and destructor for the classes.

  - Both have two versions
    1. NEW and Delete
    2. NEW[] and DELETE []
  - First two are for pointers to single objects, and last two for arrays of objects.

# Difference between static and dynamic memory allocation

## Static memory allocation

Static memory is allocated automatically by compiler when definition statements are encountered.

To make static memory allocation, the amount of the memory space to be reserved should be known at the run time.

In static memory allocation sometimes memory wastage occurs because memory is already known and it can not change.

Memory allocated at the compile time has static lifetime.

Its is faster

## Dynamic memory allocation

Dynamic memory is allocated only when there is explicit call to malloc, calloc or realloc function.

Amount of memory to be reserved can be given at the run time.

Memory wastage is avoided due to memory allocation occur at run time.

Memory allocated at run time has dynamic lifetime.

it is slower



# Meta class

- a **meta class** is a class whose instances are classes. Just as an ordinary class defines the behavior of certain objects, a meta class defines the behavior of certain classes and their instances.
- a meta class is defines as class of the class.
- A meta class hold the attributes and function which will appli to the class itself therefore it is class of class.

# Friend function

- C++ allows a way through which a function can access the private data of a class.
- Such a function need not be a class member, it may be member function of another class or may be non member function.
- This function is called FRIEND FUNCTION. The declaration should be preceded by keyword FRIEND.

Class PQr

{

Private:

.....

Public:

.....

.....

Friend void abc();

};

- The function is defined elsewhere in the program like normal function.
- Function definition does not use either keyword FRIEND or scope operator.
- Functions that are declared with FRIEND keyword are known as friend functions.

- A function can be declared as friend in number of class.
- A friend function has full access right to access the private members of class.
- Member function of one class can be friend of another class.

# Characteristics

- It is not in the scope of the class in which it has been declared as friend.
- it is not in the scope of class so it cannot be called using object of that class.
- It can be invoked like normal function ,without object.

- It can be declared either in public or private part without affecting its meaning.
- Usually, it has the objects as arguments.
- Unlike member function, it cannot access the member names directly and has to use an object name and dot membership operator with each name. like
  - A.h



# Friend Function

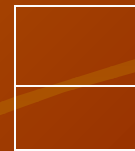
```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
friend Complex sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex sum (Complex A, Complex B)
{
Complex temp;
temp.real=A.real + B.real;
temp.imag= A.imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= sum (X,Y);
Z.putdata();
}
```

# Friend Function

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
friend Complex sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

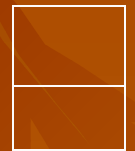
```
Complex sum (Complex A, Complex B)
{
Complex temp;
temp.real=A.real + B.real;
temp.imag= A.imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= sum (X,Y);
Z.putdata();
}
```



X



Y



Z

# Friend Function

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
friend Complex sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex sum (Complex A, Complex B)
{
Complex temp;
temp.real=A.real + B.real;
temp.imag= A.imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= sum (X,Y);
Z.putdata();
}
```

|   |
|---|
| 5 |
| 6 |

X

|   |
|---|
| 7 |
| 8 |

Y

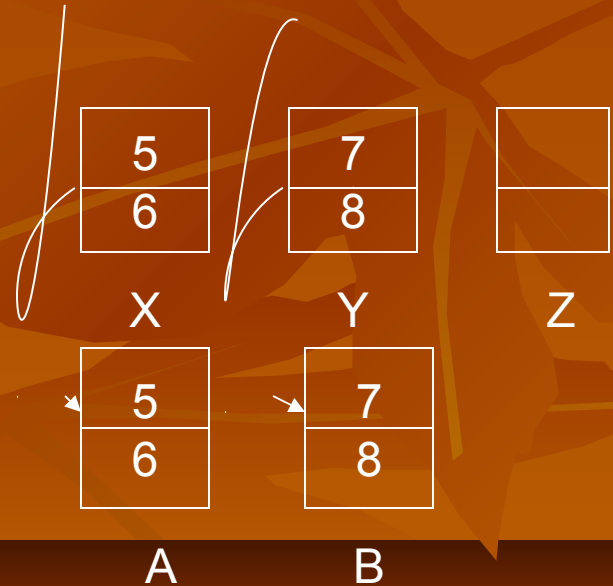
|  |
|--|
|  |
|  |

Z

# Friend Function

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
friend Complex sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
}
```

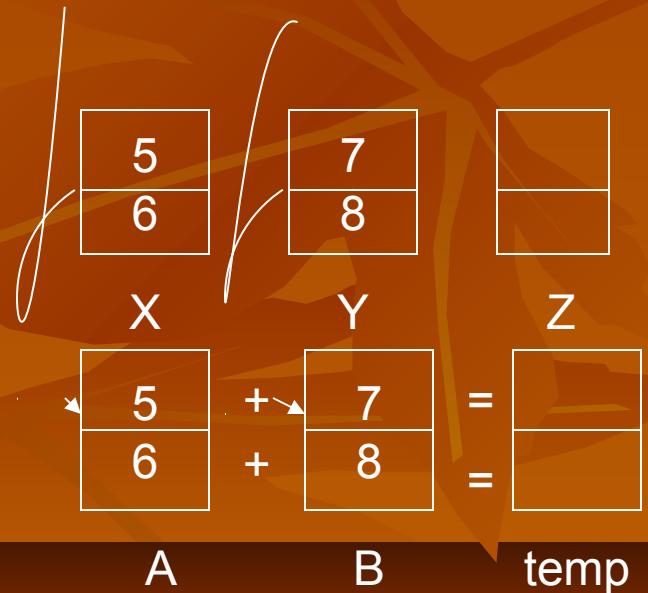
```
Complex sum (Complex A, Complex B)
{
Complex temp;
temp.real=A.real + B.real;
temp.imag= A.imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= sum (X,Y);
Z.putdata();
}
```



# Friend Function

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
friend Complex sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

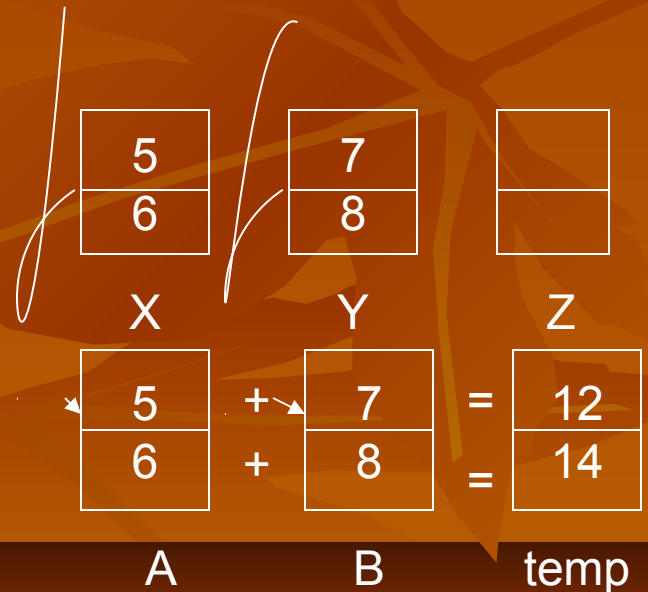
```
Complex sum (Complex A, Complex B)
{
Complex temp;
temp.real=A.real + B.real;
temp.imag= A.imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= sum (X,Y);
Z.putdata();
}
```



# Friend Function

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
friend Complex sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex sum (Complex A, Complex B)
{
Complex temp;
temp.real=A.real + B.real;
temp.imag= A.imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= sum (X,Y);
Z.putdata();
}
```

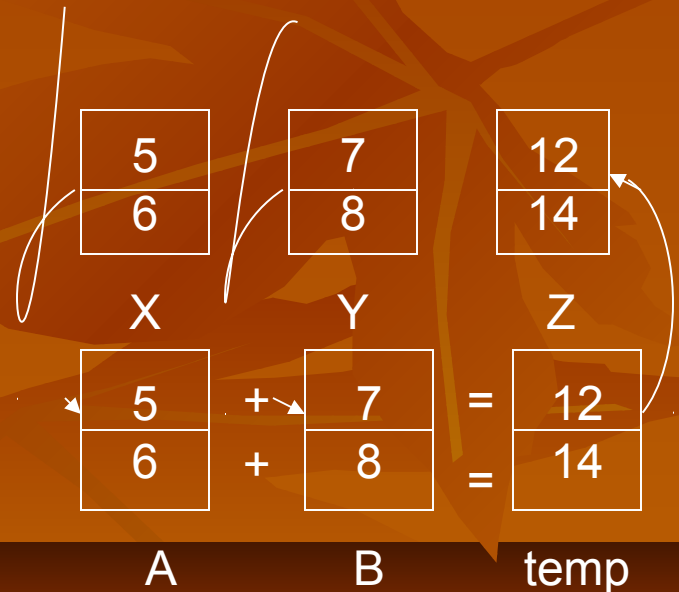




# Friend Function

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
friend Complex sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex sum (Complex A, Complex B)
{
Complex temp;
temp.real=A.real + B.real;
temp.imag= A.imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= sum (X,Y);
Z.putdata();
}
```

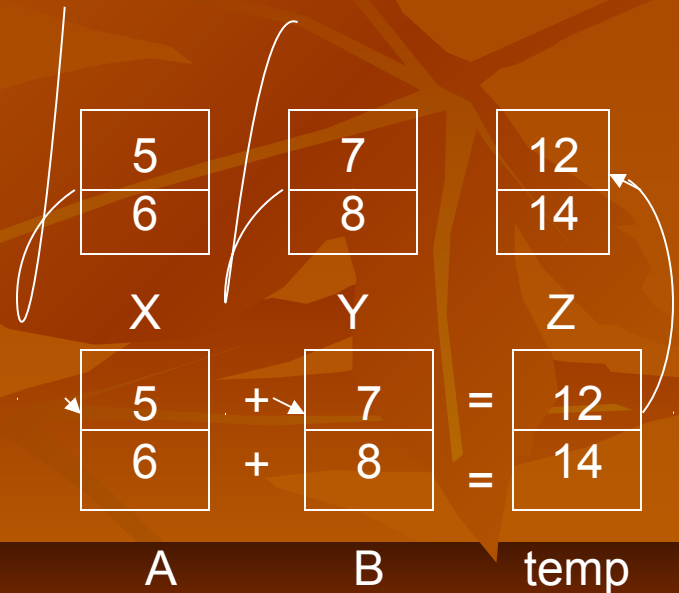


# Friend Function

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata();
void putdata();
friend Complex sum (Complex A, Complex B);
};
void Complex :: getdata()
{
cout<<"enter real part:";
cin>>real;
cout<<"enter imaginary part:";
cin>>imag;
}
void Complex :: putdata()
{
if (imag>=0)
cout<<real<<"+"<<imag<<"i";
else
cout<<real<<imag<<"i";
}
```

```
Complex sum (Complex A, Complex B)
{
Complex temp;
temp.real=A.real + B.real;
temp.imag= A.imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= sum (X,Y);
Z.putdata();
}
```

**12 + 14 i**



- We can also declare all the member functions of one class as the friend functions of another class. In this case the first class is known as FRIEND class.

- This can be specified as follows :-

```
Class z
{
.....
.....
Friend class x;
};
```

# A function friend in two classes

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

```
Class ABC;
```

```
Class XYZ
```

```
{ int x;
```

```
Public:
```

```
Void setvalue(int i)
```

```
{ x=i ; }
```

```
Friend void max(XYZ,ABC);
```

```
};
```

```
Class ABC
```

```
{ int a;
```

```
Public:
```

```
void setvalue(int i)
```

```
{ a=i ; }
```

```
Friend void max(XYZ,ABC);
```

```
};
```

```
Void max(XYZ m,ABC n)
```

```
{
```

```
If(m.x>=n.a)
```

```
Cout<<m.x;
```

```
Else
```

```
Cout<<n.a;
```

```
}
```

```
Int main ()
```

```
{
```

```
ABC abc;
```

```
abc..setvalue(10);
```

```
XYZ xyz;
```

```
Xyz.setvalue(20);
```

```
max(xyz,abc);
```

```
Return 0;
```

```
}
```

# Pass by reference

```
#include<iostream.h>
Class class_2;
Class class_1
{ int value1;
Public:
 void indata(int a)
{ value1=a; }
Void display()
{ cout<<value1<<“\n”; }
Friend void exchange(class_1 &,
 class_2 &);
};
```

```
Class class_2
{ int value2;
 public:
 void indata(int a)
{ value2=a; }
Void display()
{ cout<<value2<<“\n”; }
Friend void exchange(class_1
 &,class_2 &);
};
```

Contd...

```
Void exchange(class_1 & x,class_2
 & y)
```

```
{ int temp=x.value1;
 x.value1=y.value2;
 y.value2=temp;
}
```

```
Int main()
```

```
{
 class_1 c1;
 Class_2 c2;
```

```
C1.indata(100);
C2.indata(200);
Cout<<"values before
 exchange"<<"\n";
C1.dispaly();
C2.display();
Exchange(c1,c2);
Cout<<"values after
 exchange"<<"\n";
C1.display();
C2.display();
Return 0;
}
```



# Define a class tour in C++ with the description given below:

Private members:

tcode of type string

Noofadults of type integer

Noofkids of type integer

Kilometers of type integer

Totalfare of type float

Public members:

- A constructor to assign initial values as follows:

Tcode with the word "NULL"

Noofadults as 0

Noofkids as 0

Kilometers as 0

Totalfare as 0

- A function assignfare() which calculates and assigns the value of data member totalfare as follows:

for each adult

| Fare (Rs.) | For Kilometers |
|------------|----------------|
|------------|----------------|

|     |             |
|-----|-------------|
| 500 | $\geq 1000$ |
|-----|-------------|

|     |                          |
|-----|--------------------------|
| 300 | $< 1000 \ \& \ \geq 500$ |
|-----|--------------------------|

|     |         |
|-----|---------|
| 200 | $< 500$ |
|-----|---------|

for each kid the above fare will be 50% of the fare mentioned in the above table for example:

if kilometers is 850, noofadults = 2 and noofkids = 3

then totalfare should be calculated as

$\text{noofadults} * 300 + \text{noofkids} * 150$

I.e.  $2 * 300 + 3 * 150 = 1050$

- A function entertour() to input the values of the data members tcode, noofadults, noofkids and kilometers and invoke assignfare() function.
- A function showtour() which displays the contents of all the data members for a tour.````

```

class tour
{
char tcode[15];
int noofadults;
int noofkids;
int kilometers;
float totalfare;
public:
tour()
{
strcpy(tcode,"null");
noofadults=0;
noofkids=0;
kilometers=0;
totalfare=0;
}
void assignfare()
{
if (kilometers>=1000)
totalfare= 500 * noofadults + 250 *
noofkids;
else if (kilometers>=500)
totalfare= 300 * noofadults + 150 *
noofkids;
else
totalfare= 200 * noofadults + 100 *
noofkids;
}

```

```

void entertour()
{
cout<<"enter tcode:";
gets(tcode);
cout<<"enter noofadults:";
cin>>noofadults;
cout<<"enter noofkids:";
cin>>noofkids;
cout<<"enter kilometers=";
cin>>kilometers;
}
void showtour ()
{
cout<<"tcode="<<tcode;
cout<<"\nnumber of
adults="<<noofadults;
cout<<"\nnumber of kids="<<noofkids;
cout<<"\nkilometers="<<kilometers;
cout<<"\ntotalfare="<<totalfare;
}
};

```

# Define a class HOUSING in C++ with the following descriptions:

(4)

- private members:
  - REG\_NO integer (ranges 10-1000)
  - NAME array of characters (string)
  - TYPE character
  - COST float
- Public members:
  - function read\_data( ) to read an object of HOUSING type.
  - Function display ( ) to display the details of an object.
  - Function draw\_nos( ) to choose and display the details of 2 houses selected randomly from an array of 10 objects of type HOUSING. Use random function to generate the registration nos. to match with REG\_NO from the array.

```
class HOUSING
{
int REG_NO;
char NAME[20], TYPE;
float COST;
public:
void read_data()
{
cout<<"Enter Registration Number=";<<endl;
cin>>REG_NO;
cout<<"Enter Name=";<<endl;
gets(NAME);
cout<<"Enter Type=";<<endl;
cin>>TYPE;
cout<<"Enter Cost=";<<endl;
cin>>COST;
}
```

```
void display ()
{
cout<<"Registration No."<<REG_NO;
cout<<"\nName=";<<endl;
puts(NAME);
cout<<"\nType=";<<TYPE;
cout<<"\nCost=";<<COST;
}
void draw_nos()
{
int no1, no2;
randomize();
no1=random(1991)+10;
no2=random(1991)+10;
for (int i=0; i<10; i++)
if (arr[i].REG_NO==no1 || arr[i].REG_NO==no2)
display();
}
};
HOUSING arr[10];
```



function

# function

- Void show(); ← function declaration
- Main()
  - {
    - Show(); ← function call
    - }
  - Void show() ← function definition
  - {
  - ..... ← function body
  - .....
  - }



# Function prototype

- Introduce first in c++.
- Prototype describe the function interface ti the compiler by giving details (number and type of arguments and return type)..
- *Type function name (arguments list). ;*
- Ex:-
  - float add(int k,int g);
  - float add(int k,g); illegal
  - float add (int ,int) {name of the arguments are optional}

- In function definition arguments names are required because the arguments must be referenced inside the function ex:-

- `Float volume(int a,float b,float c);`
- `{`
  - `Float v=a * b * c;`
- `}`

- The function calling should not include type names in the argument list.

# Call by reference

- When we pass arguments by reference then the argument in the called function become alias to the actual arguments in the calling function .
- When function is working its own arguments, its works on the original arguments.
- In c++ this task is perform by making reference variable to the actual arguments.

# Call by value.

- When a function call passes arguments by value, the called function creates a new set of variable and copies the values of arguments into them, this process is known as call by value.
- Function does not have access to the actual variables in the calling program and can work on the copies of values.

# Inline function

- Inline is a function that expanded in a line when it is invoked.
- The compiler replaces the function call by its corresponding code.
- Syntax:
  - Inline return type function name
  - {
    - Function body
  - }

- Improve the execution speed.
- Reduces the memory requirement of function execution.
- All inline function must be defined before they called.
- The speed benefits of inline function diminish as the function grows in size.
- A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.



## ■ **Where inline may not work**

- For functions returning values , if a loop, switch, goto statements.
- Functions not returning values, if return exists.
- If function contain static variables.
- If inline functions are recursive
- When function call becomes small compare to function execution.

- `#include<iostream.h>`
- `Inline float add(float x , float y)`
- `{`
- `Return (x+y);`
- `}`
- `Inline float sub(float p , float q )`
- `{`
  - `Return(p-q);`
  - `}`
- `Int main()`
  - `{`
    - `Float a=12.34;`
    - `Float b=3.6`
    - `Cout<<add(a,b)<<endl;`
      - `cout<<sub(a,b)<<endl;`
  - `Return 0;`
  - `}`

# Default arguments

- A default argument is a value given in the function declaration that the compiler automatically inserts if the caller does not provide a value for that argument in the function call.
- Syntax:

```
return_type f(..., type x = default_value,...);
```

# Default arguments

- Default values are specified when the function is declared.
- We must add default values from right to left ,we can not provide a default value to a particular arguments in the middle of argument list.
- Default arguments are useful in situations where some arguments always have the same value.

# Default Arguments (Examples)

- `double pow(double x, int n=2)`
  - `// computes and returns  $x^n$`

The default value of the 2<sup>nd</sup> argument is 2.

This means that if the programmer calls `pow(x)`, the compiler will replace that call with `pow(x,2)`, returning  $x^2$

# Default Arguments (Rules)

- Once an argument has a default value, all the arguments after it must have default values.
- Once an argument is defaulted in a function call, all the remaining arguments must be defaulted.

```
int f(int x, int y=0, int n)
 // illegal
```

```
int f(int x, int y=0, int n=1)
 // legal
```



## Examples:-

- `Int mul(int I,int j=6,int l=9);` legal
  - `Int mul(int I,int j=6,int l);` illegal
  - `Int mul(int I=0,int j,int l=9);` illegal
  - `Int mul(int I=0,int j=6,int l=9);` legal
- 
- **Advantages:-**
  - We can default arguments to add new parameters to the existing functions.
  - Default arguments can be used to combine similar functions into one.

# Function overloading

- When using more than one functions with same name and with different arguments in a program is known as function overloading or function polymorphism.
- Function overloading is part of polymorphism.

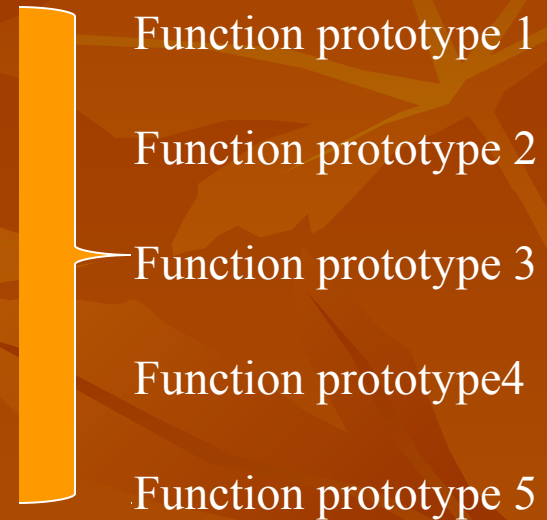
# Function overloading

- Function would perform different operations depending on the argument list in function call.
- Correct function to be invoked is determined by checking the number and type of arguments but not on return type of function.
- Examples;-
  - `Int area(int,int);`
  - `Int area(int ,float);`

# Function overloading

- Examples :-

- `Int add(int a, int b);`
- `Int add(int a, int b, int c);`
- `Double add(double x, double y);`
- `Double add(int p ,double q);`
- `Double add(double p, int q);`



- Function calls

- `Add(5,19);`
- `Add(16,7.9);`      `Add(12.4,3.5);`
- `Add (4,12,23);`      `Add(3.4,7)`

# Function overloading

- A function call first match the prototype having the same number and type of actual arguments and then calls the appropriate function for execution...

# Function overloading

- A function match includes following steps:-
  1. Compiler first try to find exact match in which the types of actual arguments are the same.
  2. If exact match not found, compiler uses the integral promotions to the actual arguments like char to int, float to double.



# Function overloading

3. When either of them fail then compiler uses built in conversion to the actual arguments and then uses the function whose match is unique.
4. If all of the steps fail then the compiler will try user defined conversions in combination with integral promotions and built in conversions to find a unique match.



Constructors

and

Destructors

# *Constructor*

- It is a member function which initializes the objects of its class.
- A constructor has:
  - (i) the same name as the class itself
  - (ii) no return type ,not even void.
- It constructs the values of data member so that it is called constructor.

- A constructor is called automatically whenever a new object of a class is created.
- You must supply the arguments to the constructor when a new object is created.
- If you do not specify a constructor, the compiler generates a default constructor for you (expects no parameters and has an empty body).

```
void main()
{
 rectangle rc(3.0, 2.0);

 rc.posn(100, 100);
 rc.draw();
 rc.move(50, 50);
 rc.draw();
}
```

- *Warning:* attempting to initialize a data member of a class explicitly in the class definition is a syntax error.

# Declaration and defination

Class complex

{

Int m,n;

Public:

complex();

};

complex :: complex ()

{

m=0;n=0;

}



- A constructor that accepts no parameters is called default constructor.

# characteristics

1. They should be declared in public section.
2. Invoked automatically when class objects are created.
3. They do not have return types, not even void and they can't return any value.
4. They cannot be inherited, though a derived class can call the base class constructors.

5. They also default arguments like other functions.
6. They implicitly call the `NEW` and `DELETE` operators when memory allocation is required.
7. Constructors can not be virtual.

# Parameterized constructors

- The constructors that can take arguments are called parameterized constructors.
- It is used when we assign different value to the data member for different object.
- We must pass the initial values as arguments to the constructors when an object is declared.

- This can be done in two ways:-

- **By calling the constructors implicitly**

- `Class_name object(arguments);`

- Ex:- `simple s(3,67);`

- This method also known as shorthand.

- **By calling the constructors explicitly**

- `Class_name object =constructor(arguments);`

- Ex:- `simple s=simple(2,67);`

- This statement create object s and passes the values 2 and 67 to it.

# Example:-

```
#include<iostream.h>
Class integer
{
 int m,n;
 public:
 integer(int,int);
 void display()
 {
 cout<<"m"<<m;
 cout<<"n"<<n; }
};
Integer::integer(int x,int y)
{
 m=x;
 n=y;
}

Int main()
{
 Integer i1(10,100);
 Integer i2=integer(33,55);
 Cout<<"object 1";
 i1.display();
 Cout<<"object 2";
 i2.display();
 Return 0;
}
```



# Notes:-

- A constructor function can also be defined as **INLINE** function.

Class integer

```
{ int m,n;
 public:
 integer (int x,int y)
{ m=x;
 n=y;
}};
```

- Parameters of a constructor can be of any type except that of the class to which it belongs.

Class A

```
{
.....
.....
Public:
A(A);
};
```

is illegal



▫ A class can accept a reference of its own class as parameter.

▫ In this case the constructor is called as copy constructor.

Class A

{

.....

.....

Public:

A(A&);

};

is valid

# Copy constructor

- When a class reference is passed as parameters in constructor then that constructor is called copy constructor.
- A copy constructor is used to declare and initialize an object from another object.
- Syntax:-
  - `Constructor _name (class_name & object);`
  - `Integer (integer &i);`

- Integer i2(i1);/integer i2=i1;
  - Define object i2 and initialize it with i1.
  - The process of initialization object through copy constructor is known as copy initialization.
- 
- A copy constructor takes a reference to an object of the same class as itself as argument.

```
▯ #include<iostream.h>
```

```
Class person
```

```
{ public:
```

```
int age;
```

```
Person(int a)
```

```
{ age = a; }
```

```
Person(person & x)
```

```
{ age=x.age;
```

```
}
```

```
};
```

```
int main()
```

```
{
```

```
Person timmy(10);
```

```
Person sally(15);
```

```
Person timmy_clone = timmy;
```

```
cout << timmy.age << " " <<
```

```
sally.age << " " <<
```

```
timmy_clone.age << endl;
```

```
timmy.age = 23;
```

```
cout << timmy.age << " " <<
```

```
sally.age << " " <<
```

```
timmy_clone.age << endl;
```

```
}
```

# Dynamic constructors

- Constructors can also be used to allocate memory while creating objects.
- This will allocate the right amount for each object when the objects are not of the same size.
- Allocation of memory to objects at the time of their construction is known as dynamic construction is known as “dynamic construction of objects”.
- The memory is allocated by NEW operator.



```

#include<iostream.h>
#include<string.h>
Class string
{
 char *name;
 int length;
Public:
 string()
 {
 length =0;
 name = newchar[length+1];
 }
 string(char *s)
 {
 length=strlen(s);
 name=new char [length+1];
 strcpy(name,s);
 }
Void display()
{ cout<<name<<“\n”;}

```

```

void join (string &a,string & b);
};
void string:: join(striing&a,string &b)
{ length =a.length+b.length;
 delete name;
 name =new char [length+1];
 strcpy(name,a.name);
 strcpy(name,b.name);
}
Int main()
{ char *first =“jon”;
string name1 (first),name2(tom),name3(jery),s1,s2;
s1.join(name1.name2);
s2.join(s1,name3);
name1.display();
name2.display();
name3.display();
s1.display();
s2.display();
return 0;
}

```

# Constructor overloading

## {multiple constructor in a class}

- When more than one function is defined in a class, is known as constructor overloading.
- Example:-

```
Class integer
{
 int m,n;
 public:
 integer()
 {
 m=0; n=0; }
 Integer (int a,int b)
 { m=a;n=b }
 Integer(integer&i)
 { m=i.m;
 n=i.n; } };
```

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

```
Class complex
```

```
{ float real,imag;
 public:
 complex(){ }
 complex(float x)
{ real=imag=x;}
 complex(float c, float d)
{ real=c; imag =
 d; }
 friend complex sum
(complex,complex);
 friend display(complex);
};
```

```
Complex sum (complex c1, complex c2)
```

```
{ complex c3;
 c3.real =c1.real +c2.real;
 c3.imag=c1.imag+c2.imag;
```

```
Return(c3);
```

```
}
```

```
Void display(complex d)
```

```
{ cout<<c.real<<"+j"<<c.imag;
}
```

```
Int main()
```

```
{
 complex a(3.4,6.7);
 complexb(2.5);
 complex c;
 c= sum(a,b);
 cou<<"a=";display(a);
 cout<<"b=";display(b);
 cout<<"c=";display(c);
```

```
}
```

- Constructors are also define with default arguments
- `Complex (float real ,float imag=0);`
- It will invoke by following way `complex c(5)`, this statement assign 5 to real and the default value already assigned to imag.
- We can also invoke it like `complex(5,3.4)`,it will assign values both real and imag means overwrite the new value to imag value.

# Dynamic initialization of objects

- Objects can be initialized dynamically, initial value of objects are provided during run time.
- Advantage of it we can provide various initialization formats by constructor overloading.

```

#include<iostream.h>
Class fixed_deposit
{
 long int pamount;
 int y;
 float r;
 float rvalue;
Public:
 fixed_deposit() { }
 fixed_deposit(long int p,int y1,float r1=0.2);
 fixed_deposit(long int p,int y1, int r1);
 void display();
};
Fixed_deposit :: fixed_deposit(long int p,int y1,
 float r1)
{
 pamount =p;
 y=y1;
 r=r1;
 rvalue=(pamount*y*r) ;
Rvalue=rvalue/100;
}

```

```

Fixed_deposit :: fixed_deposit(long int p, int
 y1,int r1)
{
 pamount=p;
 y=y1;
 r=r1;
 rvalue=pamount;
 for(int i=1;i<=y1;i++)
 rvalue =rvalue*(1+float(r)/100);}
Void fixed_deposit :: display()
{
 cout<<"\n"
<< "pricipal amount"<<pamount<<"\n"
<<"return value"<<rvalue<<"\n";
}
Int main()
{
 fixed_deposit fd1,fd2,fd3;
 long int p;int y1; float r; int R;
 Cout<<"enter amount,period,intrest rate in
 percent"<<"\n";
 Cin>>p>>y>>R;
 Fd1=fixed_deposit(p,y,R);
}

```

```
Cout<<"enter amount,period,intrest rate in
decimal " <<"\n";
Cin>>p>>y>>r;
Fd2=fixed_deposit(p,y,r);
Cout<<"enter amount and peroid" <<"\n";
Cin>>p>>y;
Fd3=fixed_deposit(p,y);
Cout<<"\ndeposit 1";
Fd1.display();
Cout<<"\n deposit 2";
Fd2.display();
Cout<<"\n deposit 3";
Fd3.display();
Return 0;
}
```



# Destructors

- A destructor is used to destroy the objects that have been created by constructor.
- It is also a member function of class whose name same as class name but preceded by tiled sign( $\sim$ ).
- It never takes any arguments nor return any value.
- It will be invoked implicitly by the compiler upon exit from the program to clean up the storage which is allocated

- The new operator is used in constructor to allocate memory and delete is used to free in destructors.

- Expl:- `~assign()`

- ```
{  
    ■ Delete p;  
}
```

```
#include<iostream.h>
Int count =0;
Class try
{ public:
    try()
    {
        count++;
    }
    Cout<<"no of objects created"<<count;
}
~try()
{
    cout<<"no of object destroyed"<<count;
    Count- -;
};
```

```
int main()
{
    cout<<"enter main";
    try t1,t2,t3,t4;
    {
        cout<<"block1";
        try t5;
    }
    {
        cout<<"block 2";
        try t6;
    }
    cout<<"again in main";
    Return 0;
}
```

Basic Concept of OOP

Ø

.

Introduction

Object oriented programming is the principle of design and development of programs using modular approach.

- Object oriented programming approach provides advantages in creation and development of software for real life application.
- The basic element of object oriented programming is the data.
- The programs are built by combining data and functions that operate on the data.
- Some of the OOP's languages are C++, Java, C #, Smalltalk, Perl, and Python.

Procedural programming

- The procedural programming focuses on **processing** of instructions in order to perform a desired computation.
- The top-down concepts to decompose main
- functions into lower level components for modular coding purpose.
- Therefore it emphasizes more on doing things like algorithms.
- This technique is used in a conventional programming language such as C and Pascal.

Object oriented programming

- Object oriented programming (OOP) is a concept that combines both the **data and the functions** that operate on that data into a single unit called the object.
- An **object is a collection of set of data** known as **member data and the functions** that operate on these data known as member function.
- OOP follows **bottom-up design** technique.
- Class is the major concept that plays important role in this approach. Class is a template that represents a group of objects which share common properties and relationships.

Differences

Procedural Programming

Large programs are divided into smaller programs known as functions

Data is not hidden and can be accessed by external functions

Follow top down approach in the program design

Data may communicate with each other through functions

Emphasize is on procedure rather than data

Object Oriented Programming

Programs are divided into objects

Data is hidden and cannot be accessed by external functions

Follows bottom-up approach in the program design

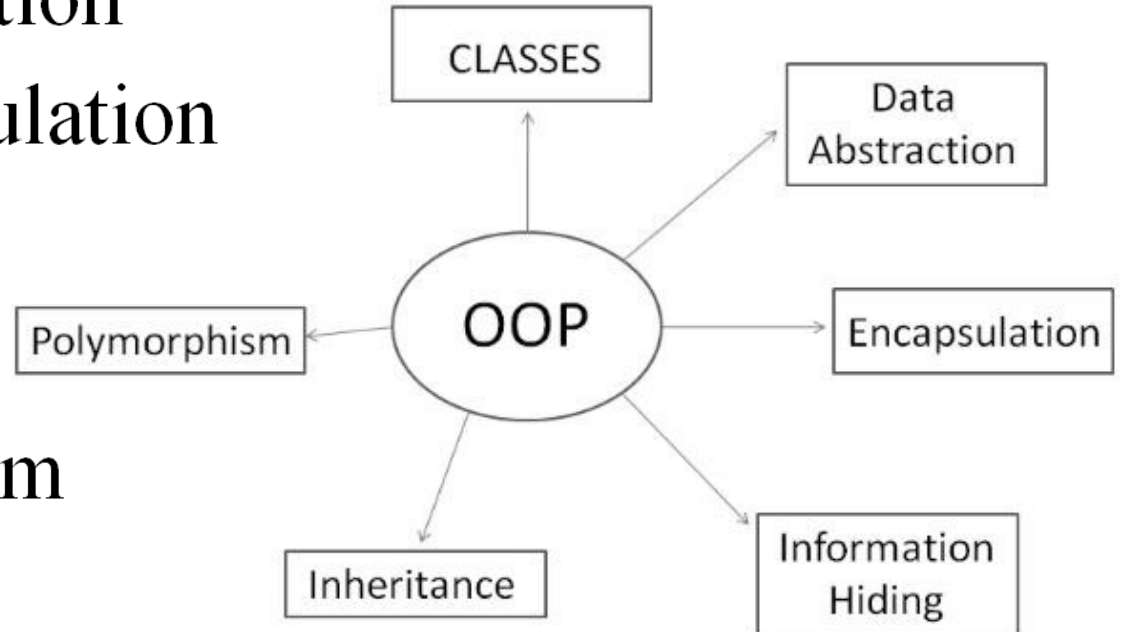
Objects may communicate with each other through functions.

Emphasize is on data rather than procedure

Basic Concepts of OOP's

The following are the major characteristics of OOP's:

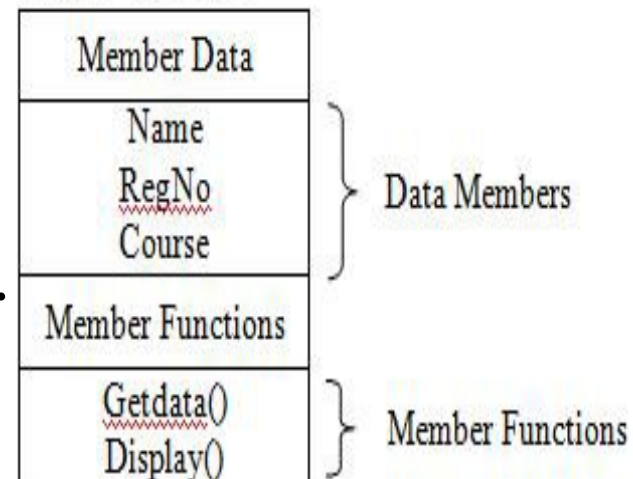
- Objects
- Class
- Data abstraction
- Data encapsulation
- Inheritance
- Overloading
- Polymorphism
- Dynamic Binding
- Message Passing



Objects

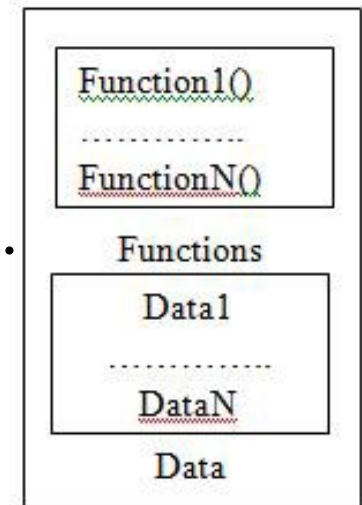
- Objects are basic building blocks for designing programs.
- *An object is a collection of data members and associated member functions.*
- An object may represent a person, place or a table of data.
- Each object is identified by a unique name. Each object must be a member of a particular class.
- Example: Apple, orange, mango are the objects of class fruit.

Object - Student



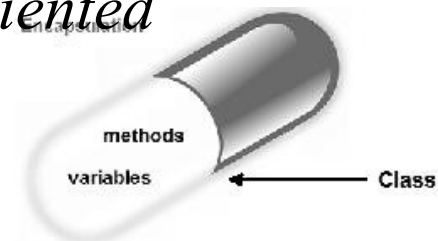
Classes

- The objects can be made user defined data types with the help of a class.
- *A class is a collection of objects that have identical properties, common behavior and shared relationship.*
- Once class is defined, any number of objects of that class is created.
- Classes are user defined data types.
A class can hold both data and functions.
- For example: Planets, sun, moon are member of class solar system.



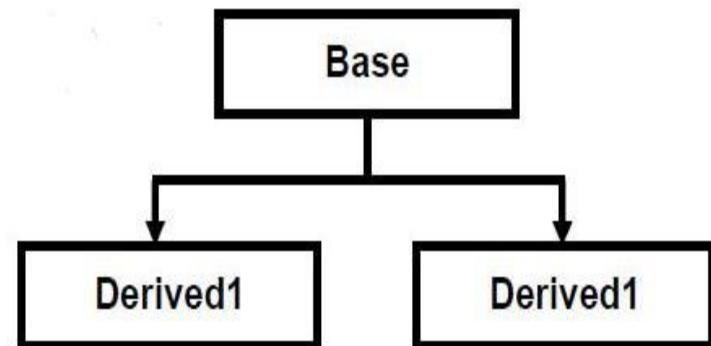
Data Abstraction, Encapsulation & Data Hiding

- **Data Abstraction:**
 - *Data Abstraction refers to the process of representing essential features without including background details or explanations.*
- **Data Encapsulation:**
 - *The wrapping of data and functions into a single unit (class) is called data encapsulation.*
 - Data encapsulation enables data hiding and information hiding.
 - *Data hiding is a method used in object oriented programming to hide information within computer code.*



Inheritance

- *Inheritance is the process by which one object can acquire and use the properties of another object.*
- The existing class is known as *base class or super class*.
- The new class is known as *derived class or sub class*.
- The derived class shares some of the properties of the base class. Therefore a code from a base class can be reused by a derived class.

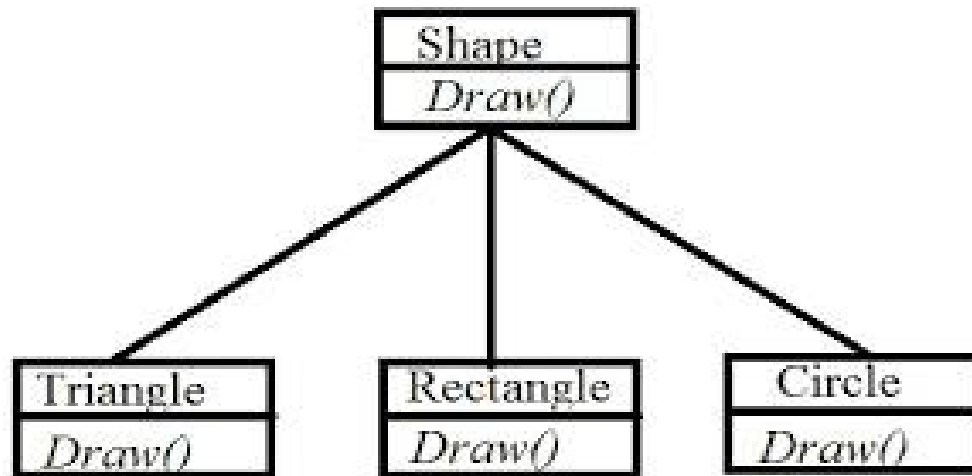


Overloading

- *Overloading allows objects to have different meaning depending upon context.*
- There are two types of overloading viz.
 - Operator Overloading
 - Function Overloading
- When an existing operator operates on new data type is called *operator overloading*.
- *Function overloading means two or more function have same name, but differ in the number of arguments or data type of arguments.*

Polymorphism

- *The ability of an operator and function to take*
- *multiple forms is known as Polymorphism.*
- The different types of polymorphism are operator
- overloading and function overloading.



Dynamic binding & Message Passing

• Dynamic binding:

- Binding is the process of connecting one program to another.
- Dynamic binding is the process of linking the procedure call to a specific sequence of code or function at run time or during the execution of the program.

• Message Passing:

- In OOP's, processing is done by sending message to objects.
- A message for an object is request for execution of procedure.
- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

Advantage of OOP's

- The programs are modularized based on the principles of **classes** and **objects**.
- Linking code & object allows related objects to share common code. This reduces **code duplication** and **code reusability**.
- Creation and implementation of OOP code is easy and reduces software development time.
- The concept of data abstraction separates object specification and object implementation.
- **Data encapsulated** along with functions. Therefore external non-member function cannot access or modify data, thus proving data security.
- Easier to develop complex software, because complexity can be minimized through inheritance.
- OOP can communicate through message passing which makes interface description with outside system very simple.

Disadvantage of OOP's

- Larger program size: OOP's typically involves more lines of code than procedural programs.
- Slower Programs: OOP's typically slower than procedure based programs, as they typically require more instructions to be executed.
- Not suitable for all types of programs.
- To convert a real world problem into an object oriented model is difficult.
- OOP's software development, debugging and testing tools are not standardized.
- Polymorphism and dynamic binding also requires processing time, due to overload of function calls during run time.

Application of OOP's

- Computer graphics applications.
- CAD/CAM software
- Object-oriented database.
- User-Interface design such as windows
- Real-time systems.
- Simulation and Modeling
- Artificial intelligence and expert systems.
- Client-Server Systems.