

Object Oriented Programming with C++

Harshita Maheshwari

Object Oriented Programming

Why OOP?

- Robust/Adaptable program are required.
- Complexity is ever increasing in software development.
- Communication gap between user and its developer
- Changing requirement during development
- Difficulty in managing software development process
- Easy user interface

Limitations of Procedural Oriented Programming

- More focus is on procedures rather than data
- Translation of user requirement into system design complex.
- Not able to secure code
- Difficult to debug or maintain the program

Need of OOP

- Main aim of OOP is to implement real-world entities.
- Increase the flexibility and maintainability of programs.
- OOP brings together data and function (behaviour) in a single location (Object) makes it easier to understand how a program works.
- Eg: c++, C#, java, python etc.

Difference between POP and OOP

Object Oriented	Procedural Oriented
Program is divided into objects	Program is divided into functions.
Bottom-up approach	Top-down approach
Inheritance property used	Inheritance is not allowed
Uses access specifier	Data is local or global (no access specifiers)
Encapsulation is used to hide the data.	Data is global
Eg:- C++, Java	Eg. C, Pascal

Object

An Object is an real world entity which has well defined structure and behaviour.

It can be tangible /intangible / conceptual.

Characteristics of Object-

- A. State
- B. Behaviour
- C. Identity
- D. Responsibility

State

State of an object is the current value of all it's attributes.

An attribute can be –

Static - remain same or do not change

Dynamic - change as per use or requirement

Eg:- Employee Object have some attributes-

Empid	}	static
Empname		
Age	}	dynamic
Salary		
Experience		

Behaviour

Behaviour is how object acts or reacts, in terms of its state changes and message passing.

Behaviour can help in retrieval of information or changes in the attributes of an object.

State changes then behaviour also changes i.e any new change in state change behaviour also.

Identity

Identity is that property of an object which distinguishes it from all other objects or uniquely identifies it.

An single attribute or group of attributes can be identity of an object.

Eg.

Employee → Empid

Student -> PRN No

Responsibility

Responsibility of an object is role it serves within the system.

It is the task for which the object is especially created.

OOP Concepts

The key features of OOP-

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

Abstraction is the process of identifying the key aspects and ignoring the rest.

Only those aspects of area are selected that are important to the current problem scenario.

Abstraction changes as per domain.

Example – Person

1. Social survey – name, age, Identity proof
2. Payroll System – name, education, job experience
3. Matrimony – name, age, height, weight, skin complexion

Encapsulation

Encapsulation is mechanism for hide the data, internal structure or implementation details of an object.

It allows to bind methods and functionality together in a single unit.

All interaction with object is through public interface of operations.

The user know only about the interface, any changes to the implementation does not affect the user.

Inheritance

Inheritance is the process by which one object can acquire the properties of another object.

“is-a” kind of relationship/ hierarchy

Broad category is formed (base class) and then sub categories are formed (derived class)

Mainly used as it supports re-usability.

Two approaches – Generalizing (bottom to up)
Specializing (top to bottom)

Polymorphism



Polymorphism

The ability of different types of related object to respond to the same message in different way.

It helps us to – design extensible software where an new object can be added to the design without rewriting existing procedures.

Containment

One object may contain another as a part or as an attribute.

It is associatively of one object with other object.

“Has-a” OR “is-part-of” relationship.

Types-

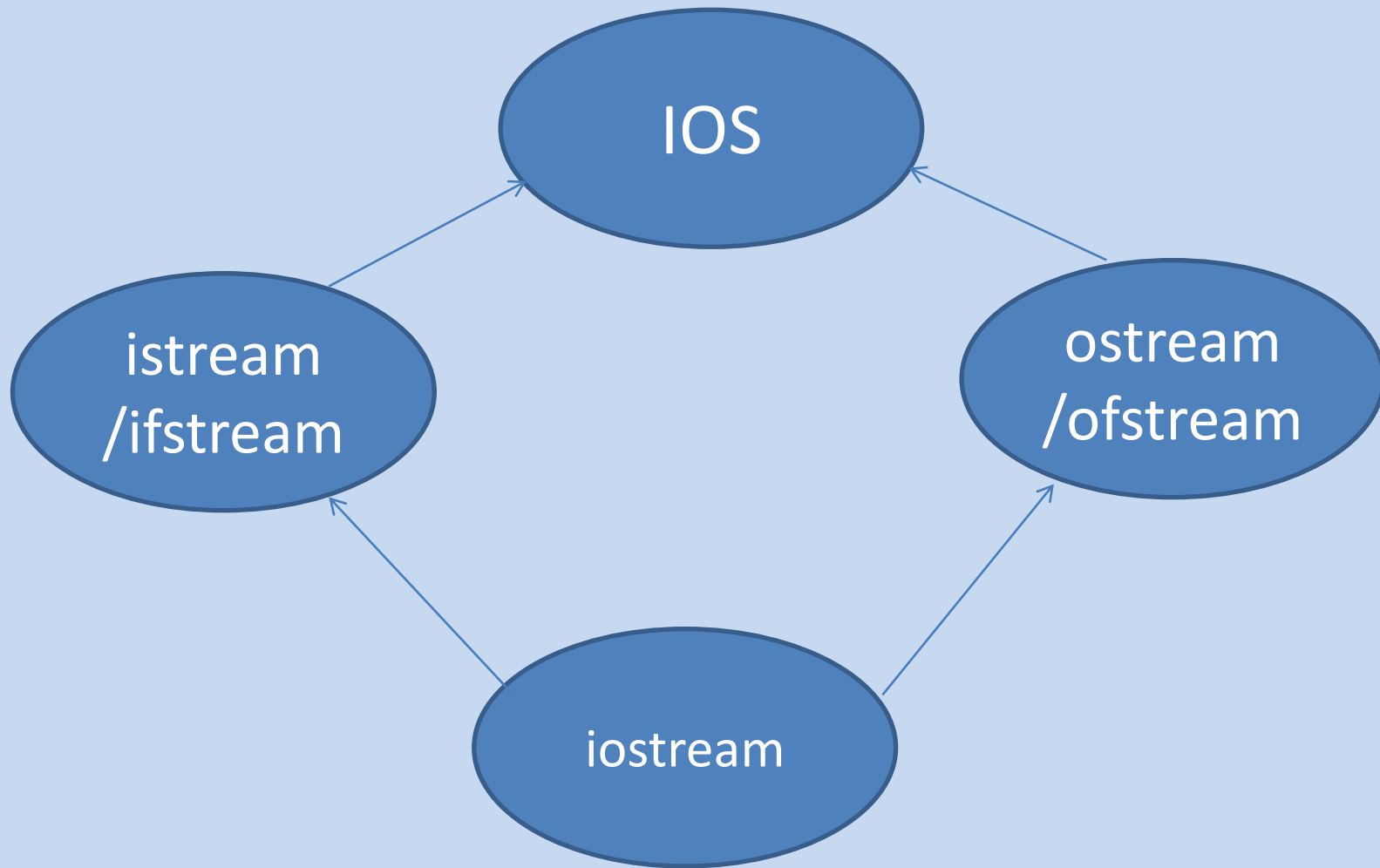
1. Aggregation (logical)
2. Composition (Physical)

Basics of CPP

CPP History

- OOP language
- CPP was invented by Bjarne Stroustrup in 1979.
- C was used as base language for inventing C++.

Using C++ standard Library



cin and cout

cin is the object of istream class.

Keyboard ----> cin >> ----> num

Eg:- cin>>num;

Cout is object of ostream class.

Screen <---- cout <---- << <----- num

Eg: cout<<"Number ="<<num<<endl;

>> = Extraction Operator

<< = Insertion Operator

Bool type

- C++ bool type can have two pre defined constant values i.e.
- true- is predefined value 1
- false - is predefined value 0

```
int main()
{
    bool flag;
    cout<<"Enter true or false "<<endl;
    cin>>flag;
    if(flag==true)
        cout<<"flag is true"<<true<<endl;
    else
        cout<<"flag is false "<<false<<endl;
    Return 0
}
```

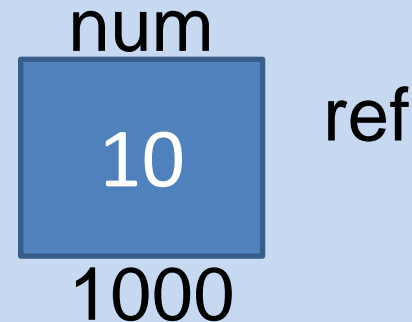
Reference variable

Reference is an nickname created for a variable.

Eg:-

```
int num=10;
```

```
int &ref =num;
```



➤ We can not create reference to reference i.e

```
int &&r=ref;
```

➤ Once created ref_name cannot assigned to another variable.

➤ References never allocate separate memory space.

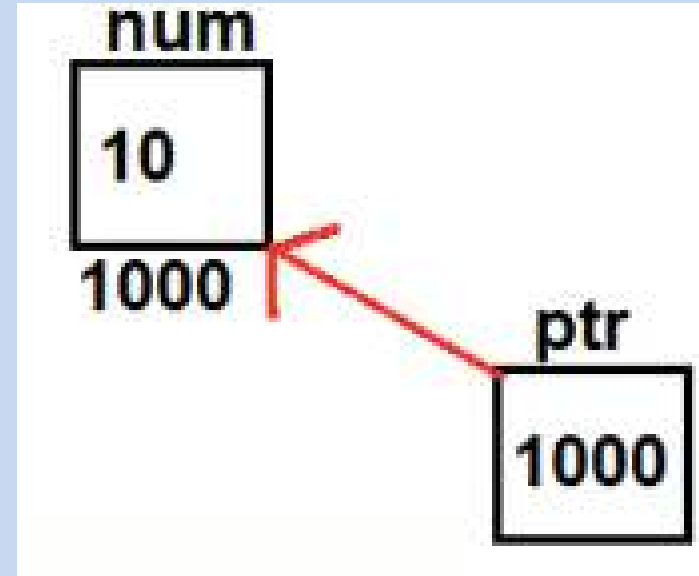
➤ References are initialized at the point of declaration.

Const with pointer

1. Pointer to constant

```
int num=10;  
const int *ptr = &num;  
or  
int const *ptr = &num;
```

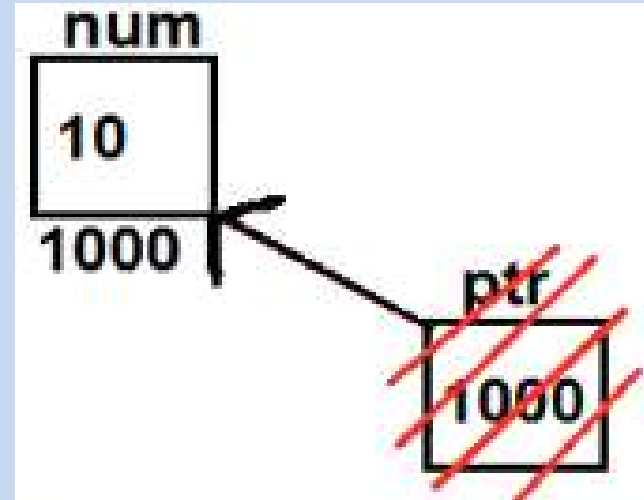
```
num=20;  
num++;  
ptr =&n2;
```



```
*ptr++;  
*ptr =30;
```

2. const pointer to int:-

```
int num=10;  
int *const ptr = &num;
```



```
num=20;  
num++;  
*ptr++;  
*ptr=30;
```



```
ptr=&n2;
```



Typecasting

Any implicit typecasting is not directly supported by compiler in C++;

Warning message for loss of precision may occur.

4 casting operator-

1. `const_cast`
2. `static_cast`
3. `dynamic_cast`
4. `reinterpret_cast`

static_cast

Syntax-

`static_cast<data_type> (expression)`

Eg-

```
double d=3.14;
```

```
int i;
```

```
i=d //using implicit typecasting
```

```
i=static_cast<int>(d)
```

Inline Functions

```
#define MAX(a,b) (a>b ?a:b)
```

Not Type Safe

Function call Overheads

```
int max(int n1,int n2)
{
    return(n1>n2?n1:n2)
}
```

```
inline int max(int n1,int n2)
{
    return (n1>n2?n1:n2);
}
```

faster as well as
type safe

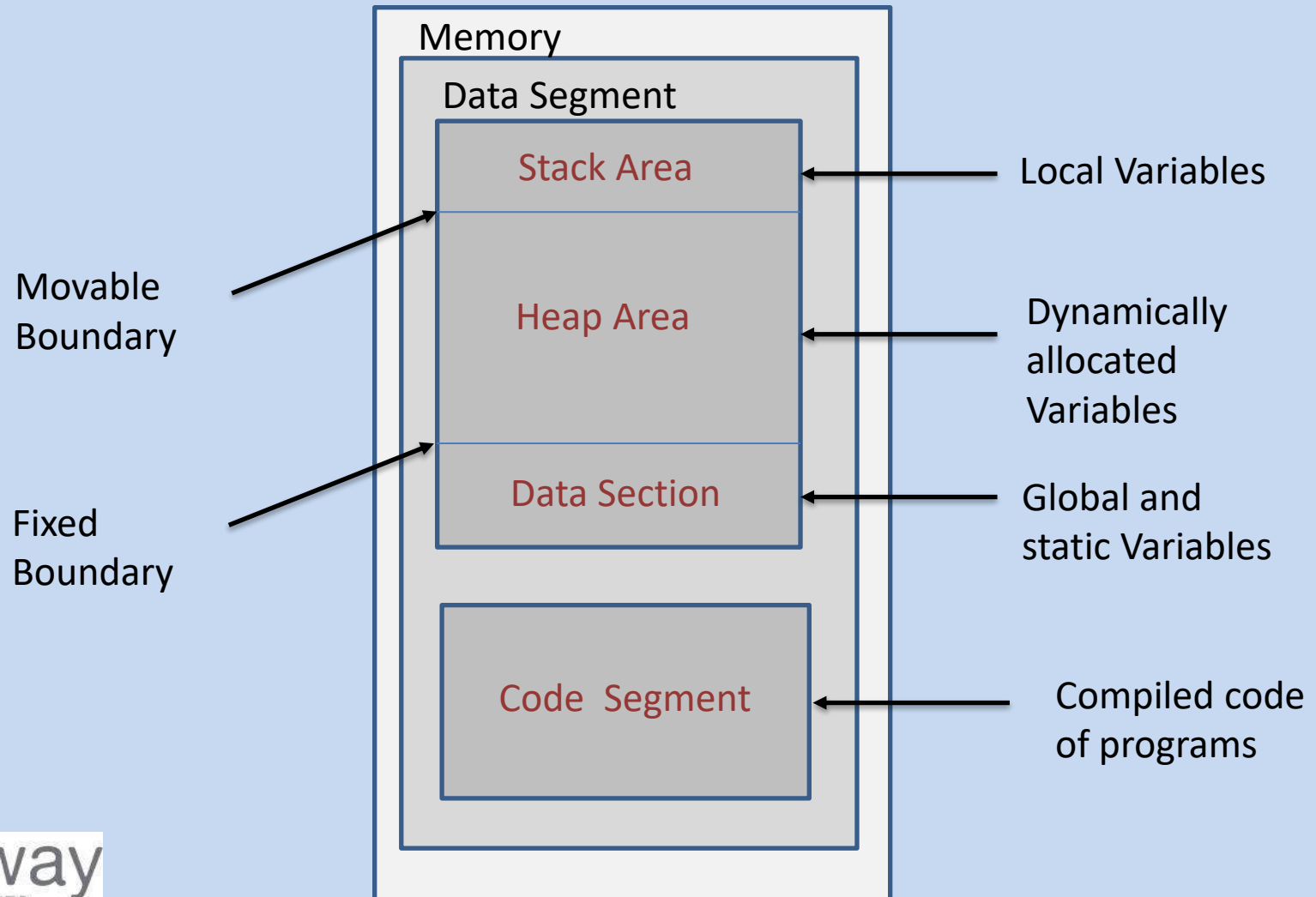
Inline Functions

While using inline function you should know

- inline is a keyword.
- For every instance of a function call the function definition gets substituted.
- inline is a request to compiler and not instruction.
- Function substitution occurs only at compiler's discretion.

Dynamic Memory Allocation

- Structure of a data segment of .exe file in memory



Dynamic Memory Allocation

new operator – allocates memory on heap

delete operator – free memory allocated by new

Syntax:- `pointer = new datatype[size];`

eg:-

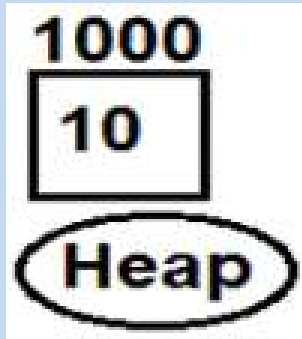
```
int *ptr = new int[size];
```

Syntax:-

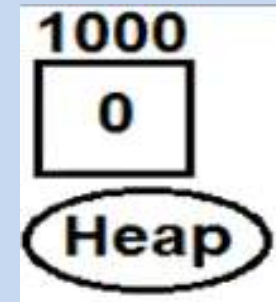
```
delete [] pointer;
```

eg: `delete []ptr;`

1. `int *ptr = new int(10)`



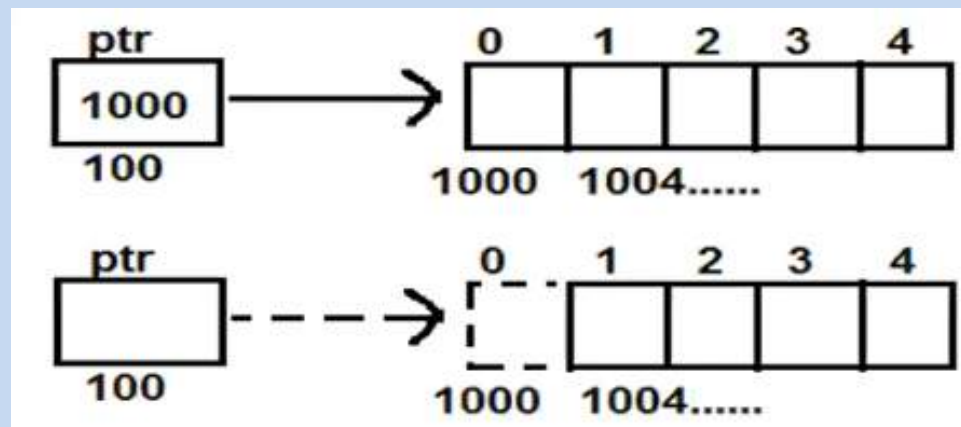
2. `int *ptr = new int;`



`delete ptr;`

3. `int *ptr = new int[n]` //n is size for array

`delete []ptr;`



Dynamic Memory Allocation

While using new/delete you should know

- sizeof() operator not required
- New returns pointer of specified data type
- Mixed use of malloc and new should be avoided.
- If [] is used with new, it should also be used with delete; otherwise it leads to memory leakage problem.

Function Overloading

Using same function name again and again within same scope.

It provides to reduce the overhead of using different function name for same algorithm.

Rules:-

- same function name within same scope (i.e same file, class)
- different number of parameter list
- different data-types and sequence of data-type
- return-type is not considered

Default Arguments to a Function

In C++ function can be declared with default values.

- C++ allows to call a function without specifying all its arguments if a function is declared with default values.
- Only trailing arguments can have default values.
- The default values must be specified in the declaration & not in the definition.

Default Arguments to a Function

Default arguments

```
void f( int x,int y, int z)
{
    cout << x;
    cout << y;
    cout << z;
}
```

```
void main (void)
{
    void f(int,int = 5,int = 0) ;
    f ( ); // error
    f ( 10); //displays10 5 0
    f(10,20,30)//10 20 30
    f( 10, ,30)//error
}
```

Classes and Object

Structure in C-

Collection of different data types in one block.

Contains variable only.

All data is global so any function or file can access it easily.

Structure in C++ -

Collection of variables and functions.

By default data is secured.

Also can explicitly give privilege to secure data i. e data will no longer be global.

```
struct Employee
{
    int eid;
    char name[20];
    void accept()
    {
        //executable logic
    }
    void display()
    {
        //executable logic
    }
};
```


class in C++

- Same a structure in c++ we can use class.
- It contains variables and functions both.
- But, in class all data is by default private.
- Class fulfill 2 major pillars i.e Abstraction and encapsulation.
- Class contains variables i.e data member and functions i. e member functions.
- and it also contains Access Specifiers.
- Using class we can map real world enitivity.

Class Syntax

```
class ClassName
```

```
{
```

```
    private:
```

```
        variable declaration;
```

```
        function declaration;
```

```
    public:
```

```
        variable declaration;
```

```
        function declaration;
```

```
};
```

If semicolon is missing compiler throws an error.

Class cDate

```
class cDate
{
private:
    int Day;
    int Month;
    int Year;

public:
    Void display();
    .....
};
```

Data members

Member Functions

```
int main()
{
    cDate d1;
    d1.display();
    return 0;
}
```

Class Components

- A class declaration consists of following components.
- Access Specifiers :- restrict access to class members
 - Private
 - Protected
 - public
- Data members
- Members Functions
- Constructor
- Destructor
- Ordinary member functions

Instantiating a class

- Object is an instance of a class.
- Object can access only public members of class.
- Memory for object is allocated.
- Whenever an object is created (3 things to remember)-
 1. Memory is allocated
 2. Constructor is called
 3. Memory is initialized

```
int main(){  
    cDate d1;  
    cDate d2(12,6,1987);  
}
```

Constructor

- Constructor is a special function with same name as it's class name.
- Constructors can not have return type, not even void.
- Constructors are implicitly called when objects are created.
- In entire lifetime of any object only once constructor is called which is at object creation.
- If your class does not have constructor, compiler provides a default constructor.

Constructor

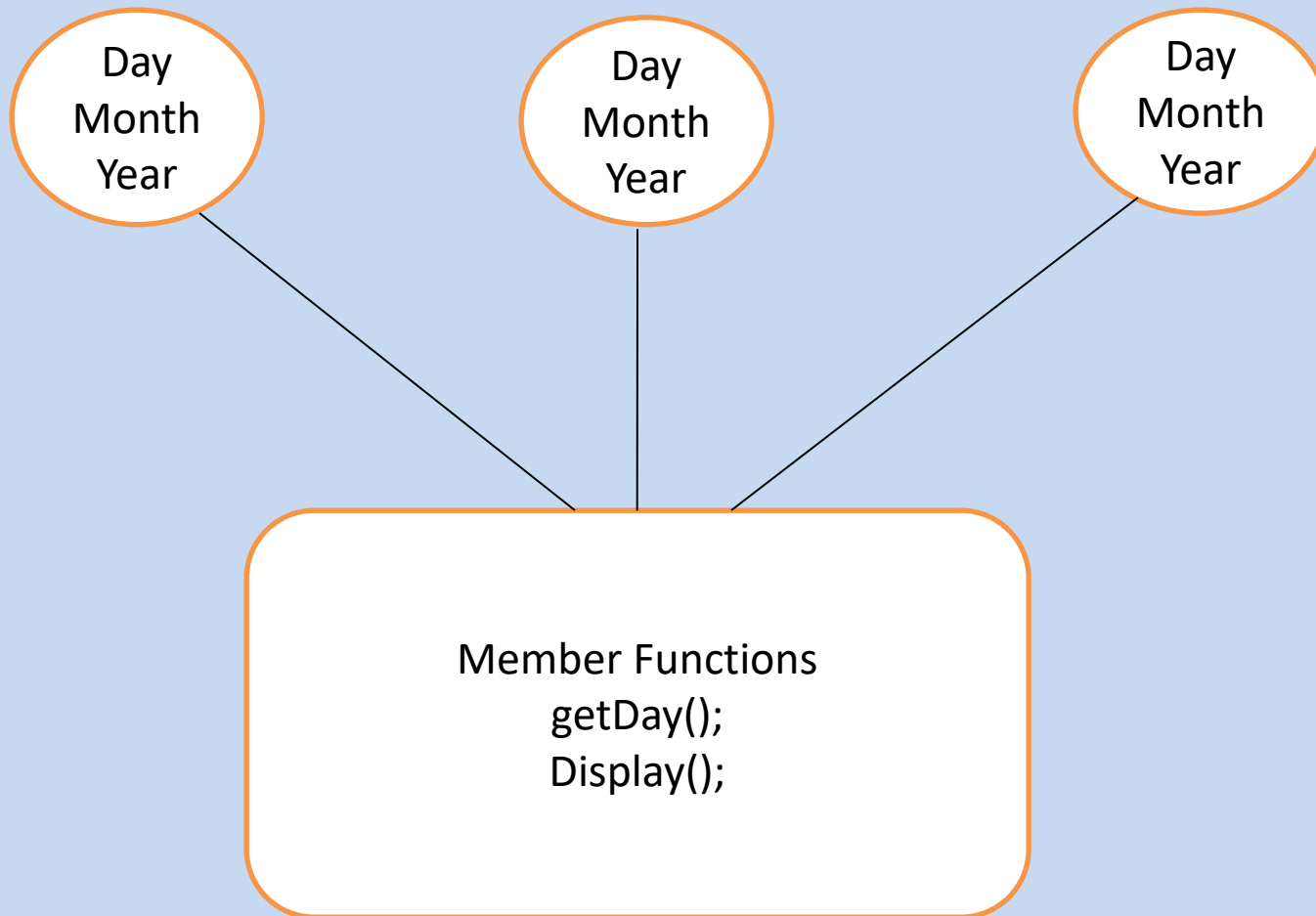
- Constructors can be overloaded.
- Type-
 1. Default(no parameter)
 2. Parameterized
 3. Copy Constructor
- Constructor overloading must follow all rules of function overloading.
- In a class you can have only one default constructor and multiple parameterized constructor.

Constructor in cDate class

```
Class cDate
{
    .....
    //no Argument constructor
    cDate()
    {
        Day=1;
        Month=1;
        Year=2000;
    }
    //Parameterized Constructor
    cDate(int d,int m,int y)
    {
        Day=d;
        Month=m;
        Year=y;
    }
};
```

```
int main()
{
    cDate d1;
    cDate d2(13,10,2016);
    return 0;
}
```


Memory Allocation to Objects



'this' pointer

- Address of an Object is passed implicitly to a member function, called on that Object.
- Every member function of class has hidden parameter : the **this** pointer.
- Pointer **this** holds the address of the Object invoking the member function.
- **this** is a keyword in C++

'this' pointer

- Pointer **this** is available only in member functions of the class.
- *Using the pointer this a member function knows on what Object it has to work.*

Using 'this' Keyword

```
int cDate:: getDay()  
{  
    return this->Day;  
} //Or return Day is same;
```

```
Void cDate::setDay()  
{  
    this->Day=d;  
}
```

```
int main()  
{  
    cDate(3,5,2000);  
    int d=d1.getDay();  
    cout<<d;  
    d1.setDay(6);  
    return 0;  
}
```

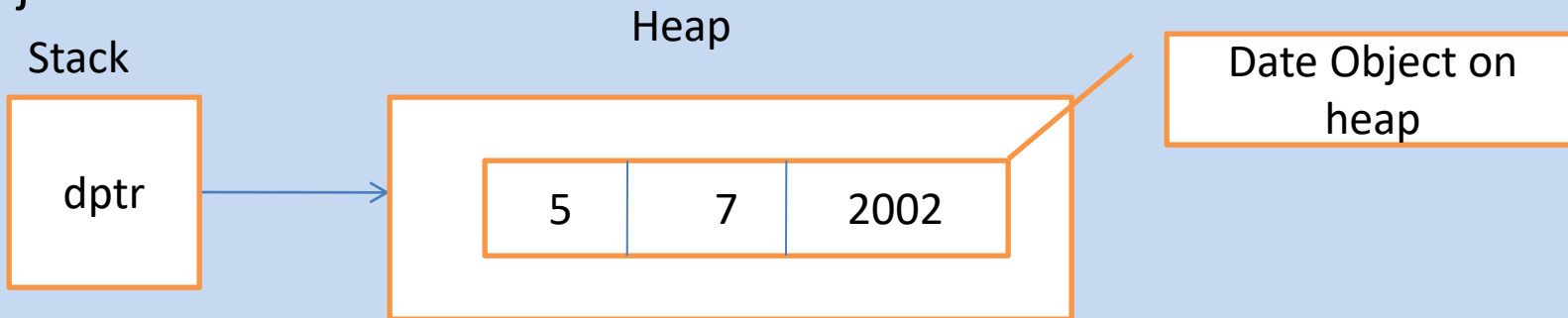
Member Functions in Class

- Facilitator function (eg. Display())
- Accessor Function (accept() , getter())
- Mutator Function (Setter())
- Helper Function (friend() . private functions)

Creating Object on Heap

```
int main()
{
    cDate *dptr=new cDate(5,7,2002);
    dptr->display();

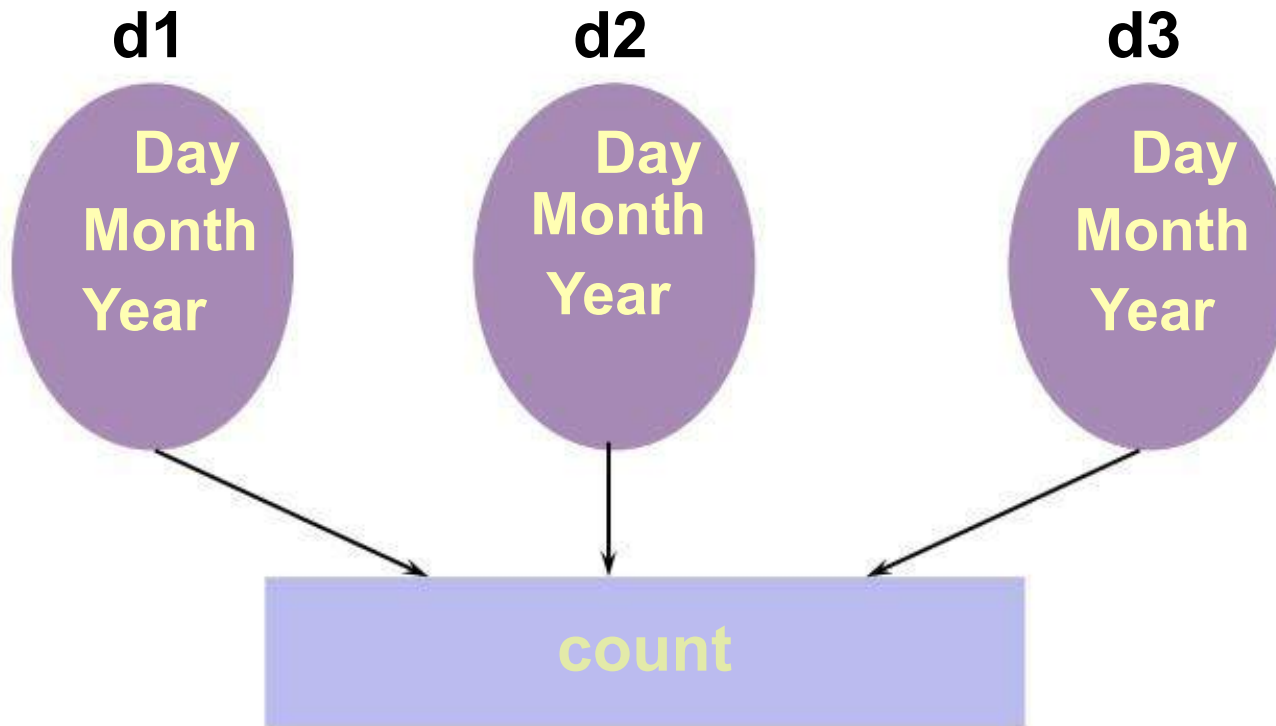
    .....
    retrun 0;
}
```



Static Data Members

- Useful when all objects of the same class must share a common item of information.
- Data to be shared by all objects is stored in static data members.
- There is single piece of storage for static data members.
- It's a class variable. Static data members could be made private, public or protected.
- Static data members are class members, they belong to class and not to any object.
- The static data member should be created and initialised before the `main()` program begins.

Objects with Static Member in Memory



Static Data Members - contd..

- As static data has single piece of memory, regardless of how many objects you have, Compiler will not allocate memory to static.
- If static data members are declared and not defined linker will report an error.
- Static data members must be defined outside the class. Only one definition for static data members is allowed.

Static Member Functions

- Static member functions can access static data members only.
- Static member function is invoked using class name.

class name :: function name()

- Pointer **this** is never passed to a static member function.
- Can be invoked before creation of any object.

Passing Object to Function

- Objects can be passed to functions in a similar manner like variables.
- If an object is “passed by value” mechanism Copy Constructor is called.
- A copy of an object is made i.e another object is created.
- To avoid creation of a copy of an object, objects should be passed by reference or by pointer.

Returning Object from function

- A function may return an object to the caller function.
- When an object is returned from a function.
- A temporary object is automatically created that holds the return value i.e temporary object is constructed by using returned object.
- Copy constructor is called for this temporary object.
- After the value has been returned to the caller function this temporary object is destroyed.

Operator Overloading

Need of Operator Overloading

- Operator overloading is feature of c++ because of which additional meanings can be given to existing operators.
- Operator Overloading is an important technique that enhances power of extensibility.
- operator is keyword in C++ which is used to implement operator overloading.
- Operator Overloading feature makes UDT's more natural & closer to built in data types.

Restrictions on Operator Overloading

You can not-

- Define new operators such as `**`.
- Change the precedence ,associativity or arity of an operator
- Meaning of operator when applied to built in types
- You can not overload
 - 1) The membership operator(`.`)
 - 2) The scope resolution operator(`::`).
 - 3) The ternary operator `?:`
 - 4) The `sizeof` operator

Format for Operator Function

```
Return_type  class_name :: operator#(argument List)
{
    .....
}
```

Where

- # is a placeholder. Substitute operator for the #
- Complex Complex::operator+(Complex &)
- Operator functions are non static member functions of class.

Operator class

Overload + operator to do addition of two Complex class objects that is

```
int main()
{
    Complex c1(3,4);
    Complex c2(1,2);
    Complex c3=c1+c2;
}
```

Internally the call is
resolved as
Complex
c3=c1.operator+(c2);

Operator class

Overload ++ operator to do pre/post increment of a Complex class objects

```
int main()
{
    Complex c1(3,4);
    Complex c2(1,2);
    Complex c3=c1++;
}
```

Overload Binary subtraction and unary '−' operator for Complex object.

Need of Friend Function

```
//part of main
```

```
{
```

```
    complex c1(2,3),c2;
```

```
    c2=c1+5;    // c1.operator +(5);
```

```
    c2=5+c1;    // ?? Will it work.
```

```
}
```

For the Statement `c2=5+c1` ;operator + can not be overloaded as member function

Need of Friend Function

To do so ,it has to be overloaded as non member function.

- Friend functions are not members of a class.(i.e non member function)
- They are normal global functions and hence do not implicitly receive this pointer.
- Friend Function can access private members of a class.

Friend Function

- Friend Function is written as other normal function
`return_type functionName(argument_list) {}`
- Only preceded friend keyword before function declaration
- Friend functions can be declare in private or public section of class
- When we are working with different 2 types of objects.
- When LHS operant is not a class type.
- Using friend function we can access private members of class outside the class.

Using Friend complex class

```
class complex
```

```
{
```

```
    friend comeplx operator+(int,const complex&);
```

```
};
```

```
complex operator+(int num,const complex &c)
```

```
{
```

```
    complex temp;
```

```
    temp.real=num+c.real;
```

```
    temp.img=num+c.img;
```

```
    return temp;
```

```
int main()
```

```
{
```

```
    complex c1(3,5),c2;
```

```
    c2=5+c1;
```

```
}
```

String Class and Copy Constructor

String class

```
class String
{
    int length;
    char *sptr;
public:
    String();
    void display();
    .....
};
```


No-argument constructor

Class String

```
{  
    public:  
    String()  
    {  
        length=0;  
        sptr=new char[length+1];  
        *sptr='\0';  
    }  
};
```

int main()

```
{  
    String s1;  
    return 0;  
}
```

Parameterized constructor

Class String

```
{ public:
    String(char ch,int len)
    {
        length=len;
        sptr=new char[length+1];
        for(int i=0;i<length;i++)
            sptr[i]=ch;
        *sptr='\0';
    }
};
```

int main()

```
{
    String s1('S',5);
    return 0;
}
```

Parameterized constructor

Class String

```
{  
    ...  
    public:  
    String(const char *s)  
    {  
        length=strlen(s);  
        sptr=new char[length+1];  
        strcpy(sptr,s);  
    }  
};
```

int main()

```
{  
    String s1("INFOWAY");  
    return 0;  
}
```

Destructor

- Destructor is also a special function with the same name as class name prefixed by ~ character. Eg ~Complex() ; or ~String();
- No need to specify return type or parameters to destructor function.
- Destructor cannot be overloaded. Therefore a class can have only one destructor.
- Destructor is implicitly called whenever an object ceases to exist.

Destructor

- Destructor function de-initializes the objects when they are destroyed.
- A destructor is automatically invoked when object goes out of scope or when the memory allocated to object is de-allocated using the delete operator.
- If a class contains pointer variable then it is mandatory on programmers part to write a destructor otherwise there is problem of memory leakage.

Destructor

Class String

```
{  
  
    .....  
    public:  
    ~String()  
    {  
        if(sptr)  
        {  
            delete [] sptr;  
            sptr=NULL;  
        }  
    }  
  
};
```

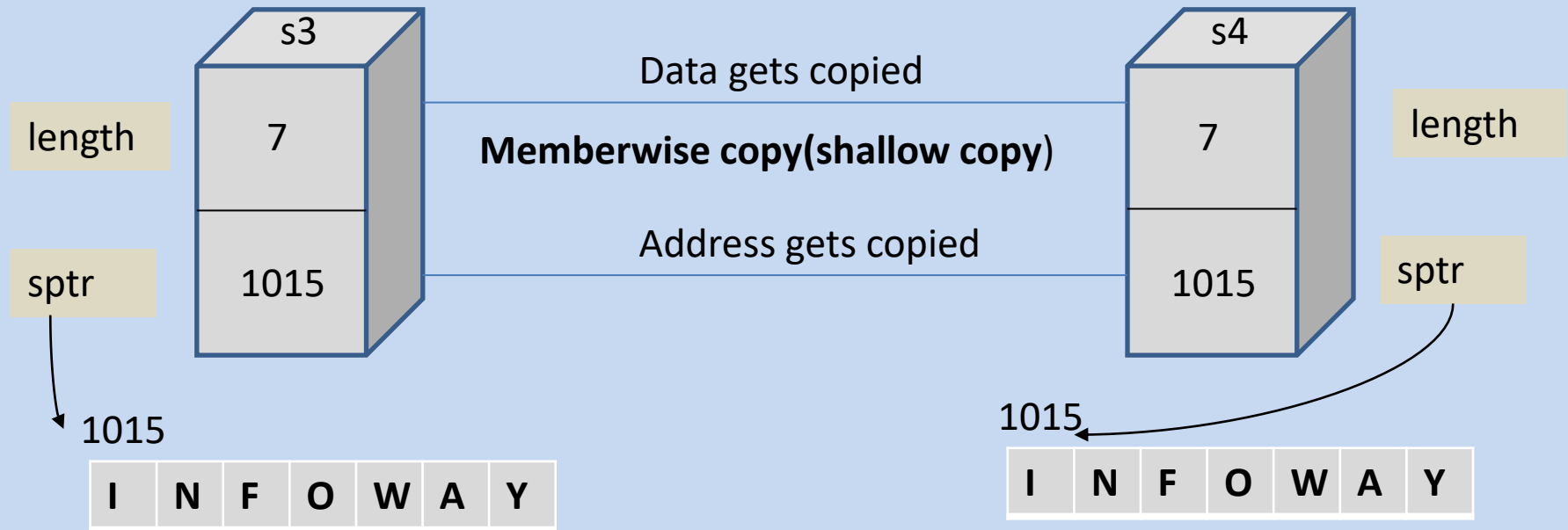
Object Creation/Destruction

- Sequence of Object creation
 - 1) Memory is allocated
 - 2) Constructor is called
 - 3) Memory is initialized

- Sequence for Object Destruction
 - 1) Destructor is called
 - 2) If memory is allocated dynamically it is freed
 - 3) Memory allocated to objects is deallocate only when object goes out of scope.

Need of Copy Constructor

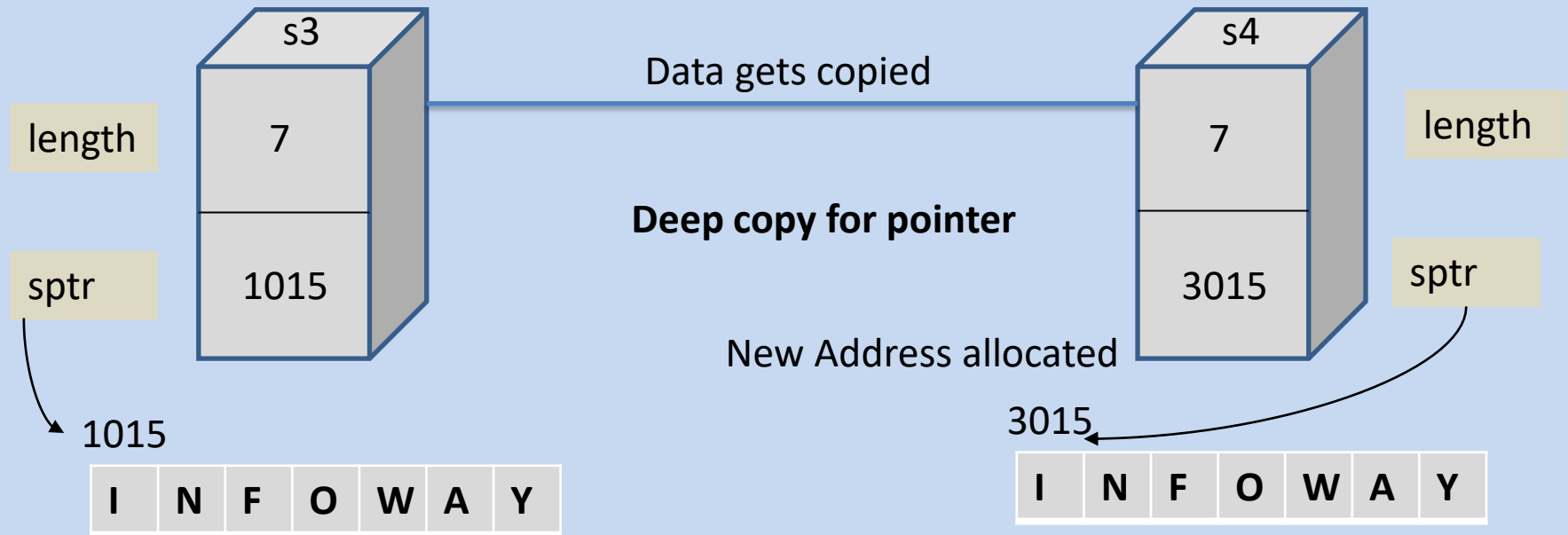
String s4(s3); //To create s4 as copy of s3



If any one of the object goes out of scope .Pointer data member of other object becomes “**Dangling Pointer**”;

Need of Copy Constructor

String s4(s3); //To create s4 as copy of s3



Copy constructor allocates separate memory on heap for string to be copied.

Copy Constructor for String class

Class String

```
{  
  
    .....  
    public:  
    String(const String &s)  
    {  
  
        this->length=s.length;  
        this->sptr=new char[this->length+1];  
        strcpy(this->sptr,s.sptr);  
  
    }
```

int main()

```
{  
    String s3("INFOWAY");  
    String s4(s3);  
}
```

Copy Constructor –Points to note

While creating copy of an Object

- Compiler Provides default copy constructor which does member wise copy.
- If a class contains one of it's data member as a pointer type ,it is mandatory on programmers part to write user defined copy constructor.
- User defined copy constructor should take care of dangling pointer situation.
- To avoid infinite recursion, pass the parameters by reference to the copy constructor.

Containment and Inheritance

Containment

- Containment represents 'has-a' or 'is a part of' relationships.
- It depicts how one object is part of another object.
For example: engines, wheels, cd-players etc are parts of car.
- Container relationship enables reusability of code.
Engine is also used in an airplane.
Pan card is used for opening bank account as well as for IT returns.

Containment example

```
class Employee
{
    int empid;
    int sal;
    String name;
    Date bdate;

public:
    Employee(int e,int s,char *name,int d,int m,int y);

};
```

Using Constructor Syntax

```
Employee::Employee(int e,int s,char *nm,int d,int m,int y)
{
    empid=e;
    salary=s;
    name=String(nm);
    bdate=Date(d,m,y);
}
```

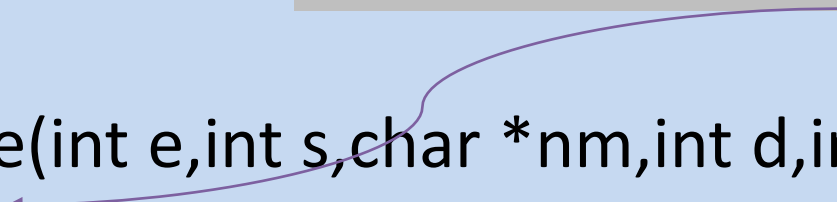
Execution sequence

- Default constructor for class String
- Default constructor for class Date
- Constructor for class Employee through which
 - Parameterized constructor for class String
 - Parameterized constructor for class Date.
- Total no of constructors invoked =5

Using member initializer list

Member Initializer List

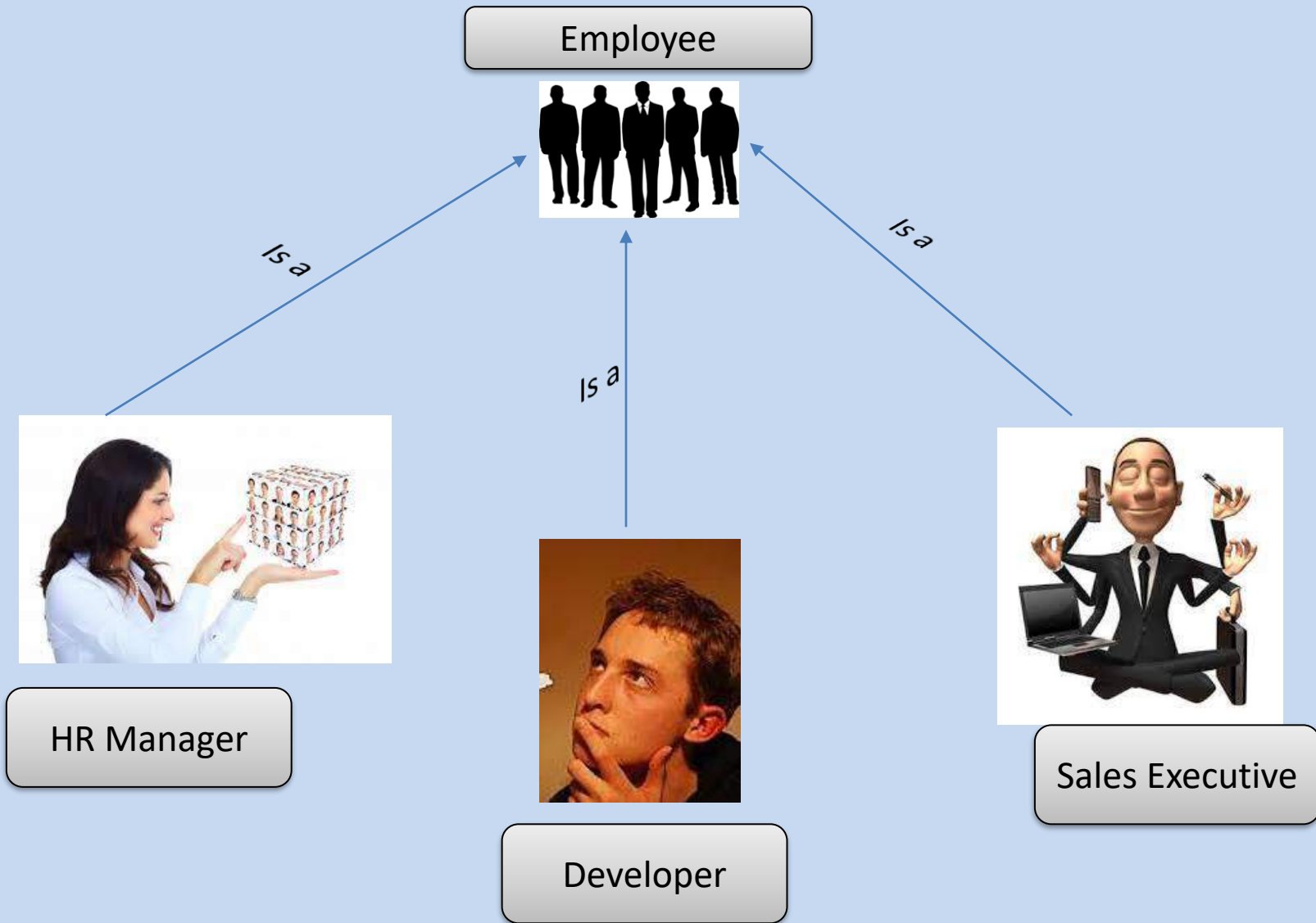
```
Employee::Employee(int e,int s,char *nm,int d,int m,int y):  
    name(nm),bdate(d,m,y)  
{  
    empid=e;  
    salary=s;  
}
```



Execution sequence

- While using Member initializer list ,constructors for contained objects are invoked first and then constructors for container objects.
- Parameterized constructors called directly.
- Constructor for class Employee through which
 - Parameterized constructor for class String
 - Parameterized constructor for class Date.
- Total no of constructors invoked =3

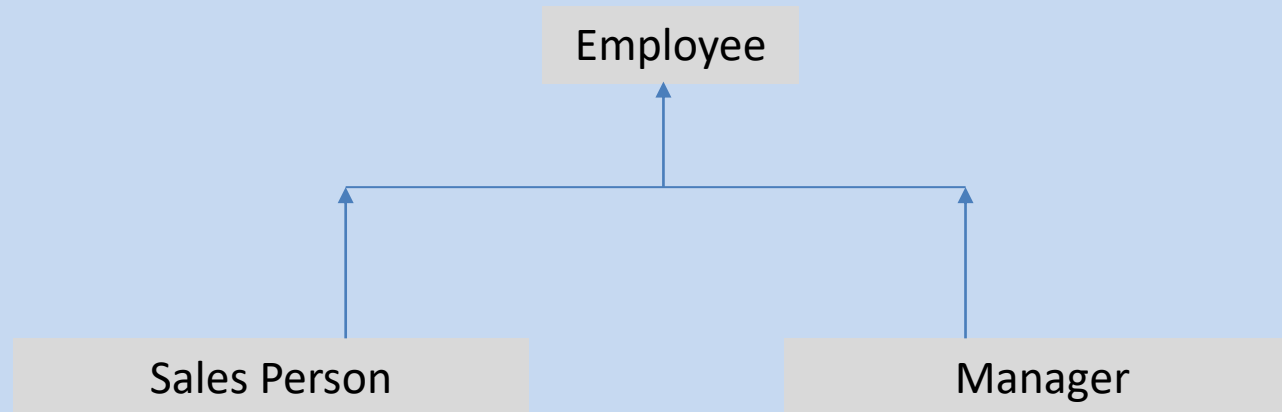
Relationship



Inheritance

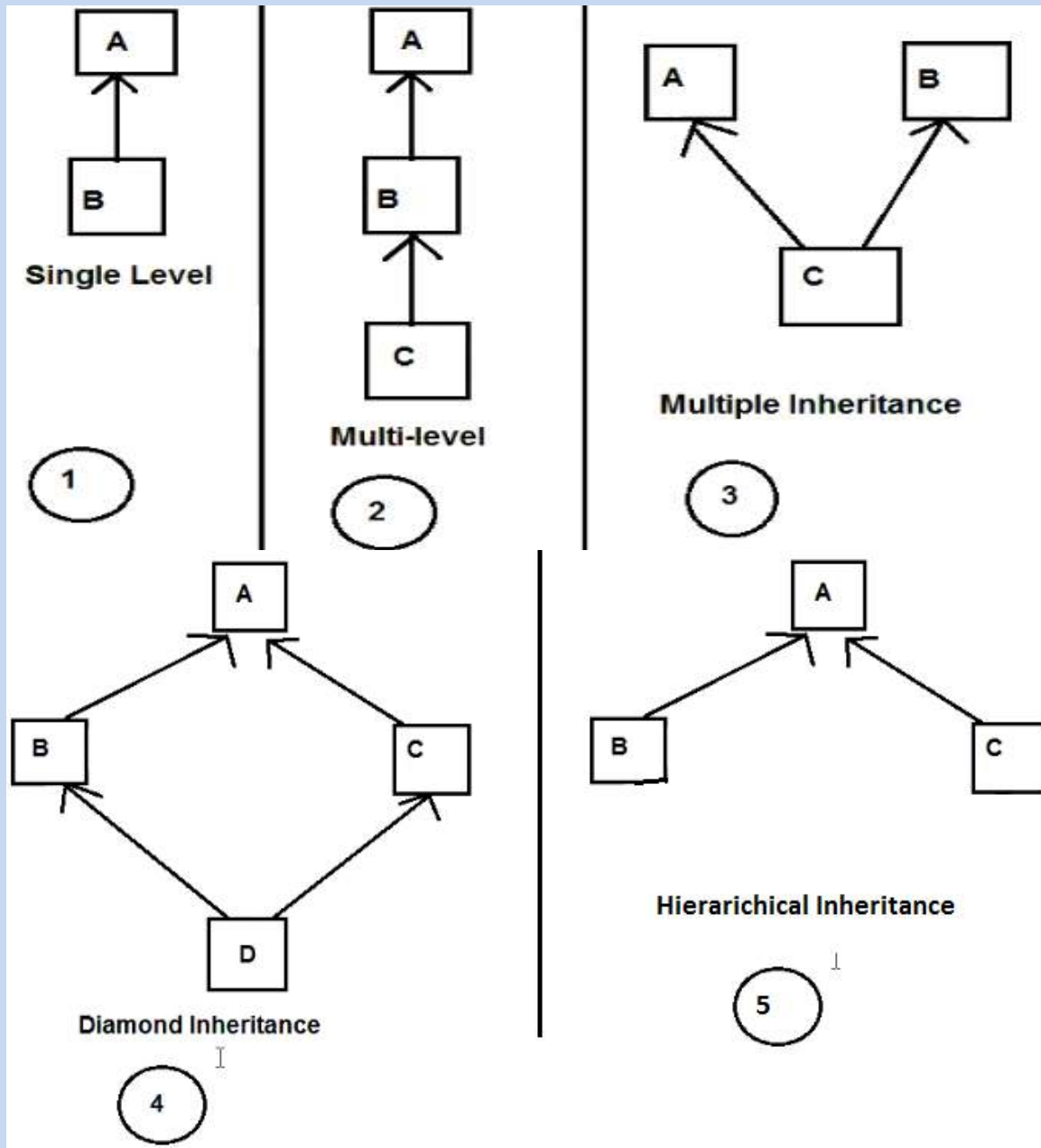
- One of the key concepts of object oriented programming approach.
- Allows creation of hierarchical classification.
- Advantages are-
 - Reusability
 - Extensibility

Base and Derived class



- This is 'is a' kind of relationship.
- More than one class can inherit attributes from a single base class.
- Derived class can be base class to another class.

Types of Inheritance



Inheritance syntax

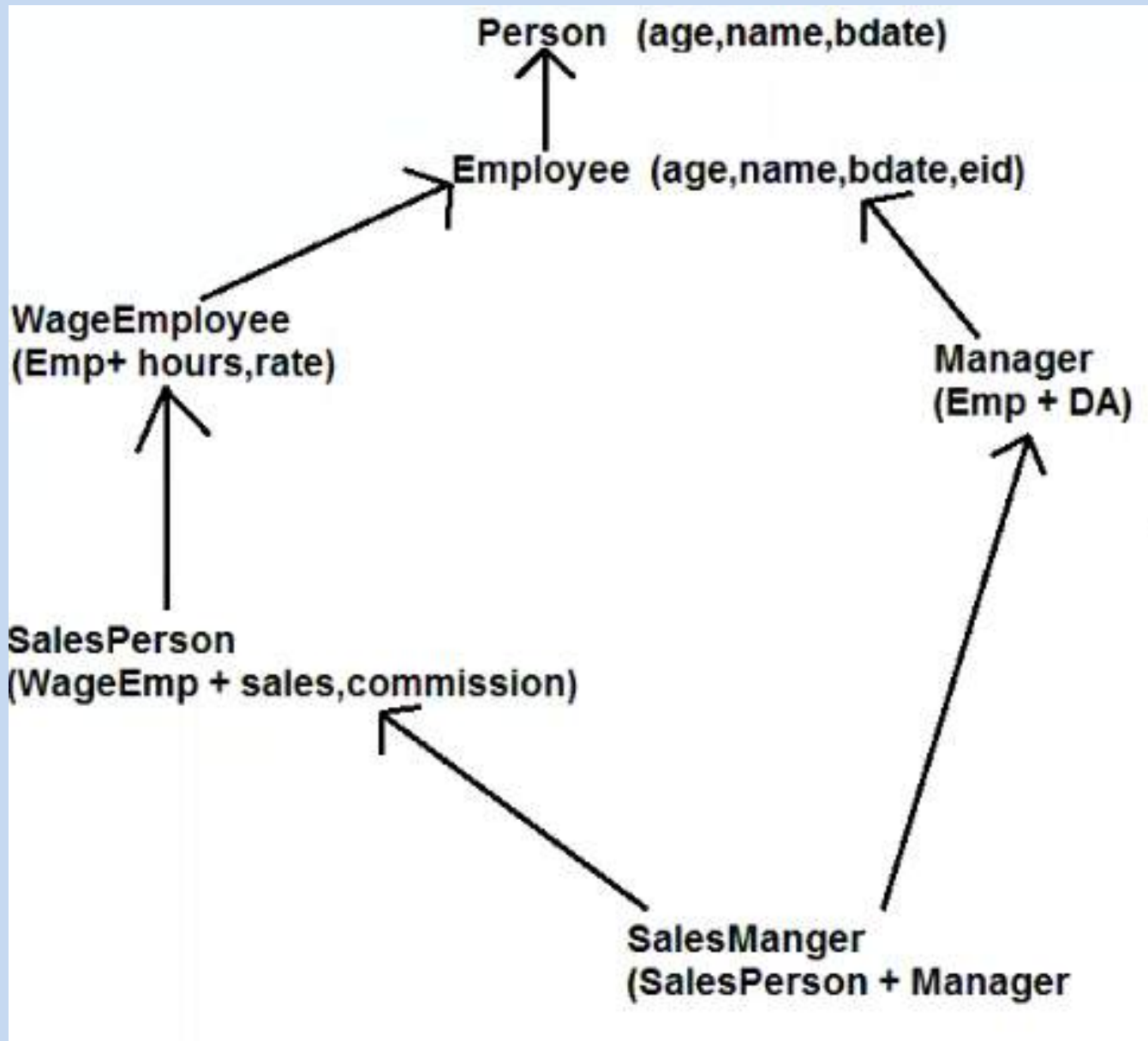
```
class base_class  
{  
    //code for base class  
};
```

```
Class derived_class:access base_class name  
{  
    //code for derived class  
  
};
```

Access can be public or private or protected.

Access Specifiers

class A	class B: private A	class B: protected A	class B: public A
private: a	private: x b c	private: x	private: x
protected: b	protected: y	protected: y b c	protected: y b
public: c	public: z	public: z	public: z c



Class SalesPerson

```
class SalesPerson:public Employee
```

```
{
```

```
    float sales;
```

```
    float comm;
```

```
    public:
```

```
    void display();
```

```
    void compute_salary();
```

Base initializer list



```
};
```

```
SalesPerson::SalesPerson(int e,int sal,const char *nm,int d,int m,int  
y,float s,float c):Employee(e,sal,nm,d,m,y)
```

```
{
```

```
    sale=s;
```

```
    comm=c;
```

Derived class constructor and destructor

- Constructors are called in the sequence of Base->derived
- When Object of derived class is created the sequence of constructor calling is

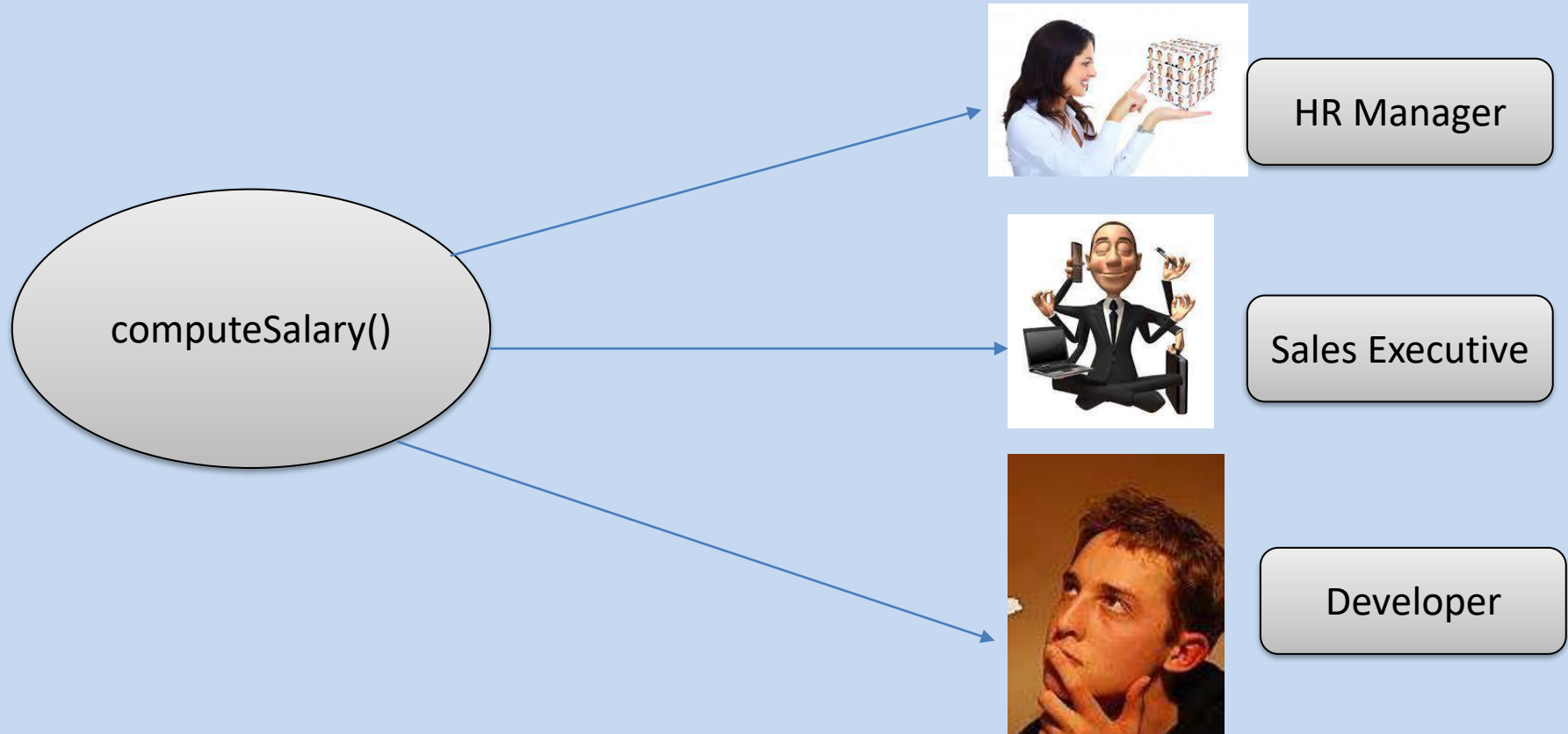
String->Date->Employee->SalesPerson

- Destructors are called in the sequence of Derived -> base

SalesPerson->Employee->Date->String

Relationship

- Ability of different related objects to respond to the same message in different ways is called Polymorphism.



Compile time and Run time binding

- Binding is an association of function call to an object.
- Compile time binding
 - The binding of member function call with an object at compile time.
 - Also called as dynamic binding or late binding.
- Runtime binding
 - The binding of function call with an object at run time.
 - Also called as dynamic binding or late binding
 - Achieved using virtual functions and inheritance.

Generic pointer

```
int main()
{
    Employee e(...), *emp;
    SalesPerson s1(...);
    emp=&e; }
    emp=&s1;}
}
```

Base class or generic pointer can point to objects of derived class.

- Assigning derived class objects to base class pointer or reference is always safe. But assigning derived class object to base class objects leads to “Object Slicing”.

Type casting

```
int main()
{
    Employee e(...), *emp;
    SalesPerson s1(...);
    emp=&s1;
    ((SalesPerson *)emp)->computeSalary()
}
```

- Based on which computeSalary() should be called ,Employee pointer is type casted to a particular type.
- Compiler always checks static data types.

Virtual Functions

- To implement late binding ,the function is declare with the keyword virtual in the base class.
- Virtual function is a member function of a class.
- Virtual functions can be redefined in the derived class as per the design of the class.
- Also consider virtual by compiler.

Generic pointer

- Declare computeSalary() as virtual in Employee class.

```
int main()
{
    Employee e(...), *emp;
    SalesPerson s1(...);
    emp=&s1;
    emp->computeSalary()
}
```

- Dynamic data type of generic pointer will govern method invocation.

Virtual Functions....points to note

- It should be non static member function of base class.
- Can not be used as friend function.
- Constructor can not be virtual but destructors can be.
- If Function is declare virtual in base class then it is treated virtual in derived class too, even if virtual keyword is not used explicitly .

Overloading vs Overriding

	Overloading	Overriding
Scope	Within same class	In the inherited classes.
Purpose	Method names need not be remembered	Message is same but implementation is specific to particular class.
Signature	Different for each function	Has to be same in all derived classes.
Return Type	can be same or different for each function but it is not consider	Return type also needs to be same .

Pure Virtual Functions

- Virtual function without any executable code .
- Declare as follows,let say in Employee

```
virtual float computeSalary()=0;
```

- A class containing at least one pure virtual function is known as abstract class.

Types of classes

- Concrete class
- Abstract class
- Pure abstract
- Polymorphic class

Abstract class

- An object of an abstract class can not be created.
- Pointer or reference of abstract class can be created.
- Therefore ,abstract class supports Runtime polymorphism.
- Pure virtual functions must be overridden in derived classes; Otherwise derived classes also treated as abstract.

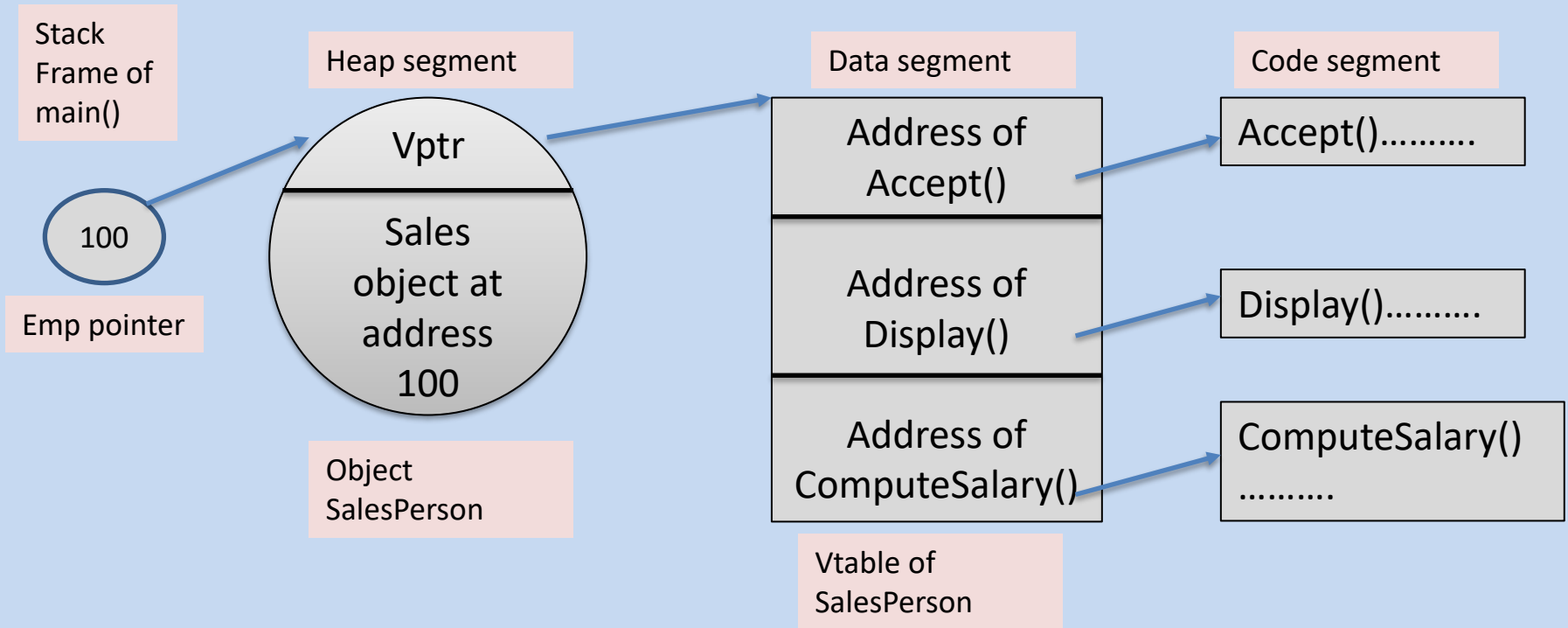
How virtual function work?

- For every polymorphic class compiler implicitly adds data member referred as vptr to every object.
- Virtual pointer is a hidden pointer
- Vptr is a pointer to static table of function pointers called Virtual table(vtable).

How virtual function work?

- Vptr is initialized to the starting address of the vtable in the constructor.
- Virtual table is an array of function pointers that contains pointers to all the virtual functions in the class.
- Virtual table is static member of a class since all objects of that class need reference to virtual table.

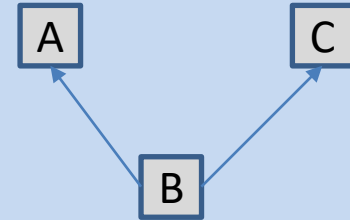
How virtual function work?



Problems with multiple inheritance

```
class B:public A,public C{ .....
```

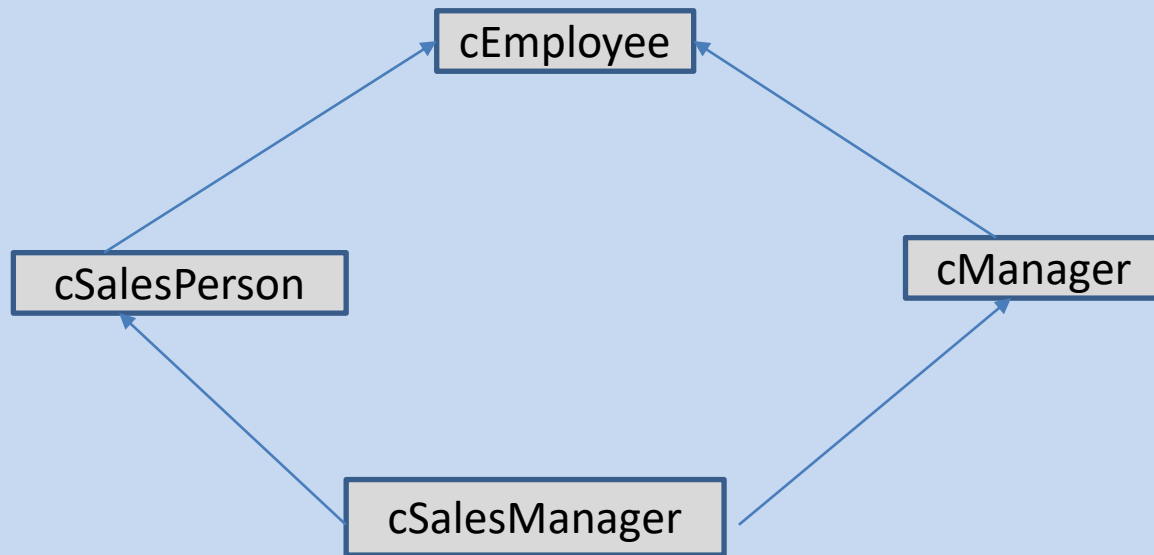
```
B obj;  
Obj.func();
```



- If multiple base class contains function with same name. Compiler throws ambiguity error ,as obj can see two copies of same functions from two classes.
- To resolve this ambiguity by two ways
 - Obj.A::func()
 - Override func in B class

Diamond Inheritance

- When a class inherits two classes, each of which inherits from a single base class, it leads to diamond inheritance pattern

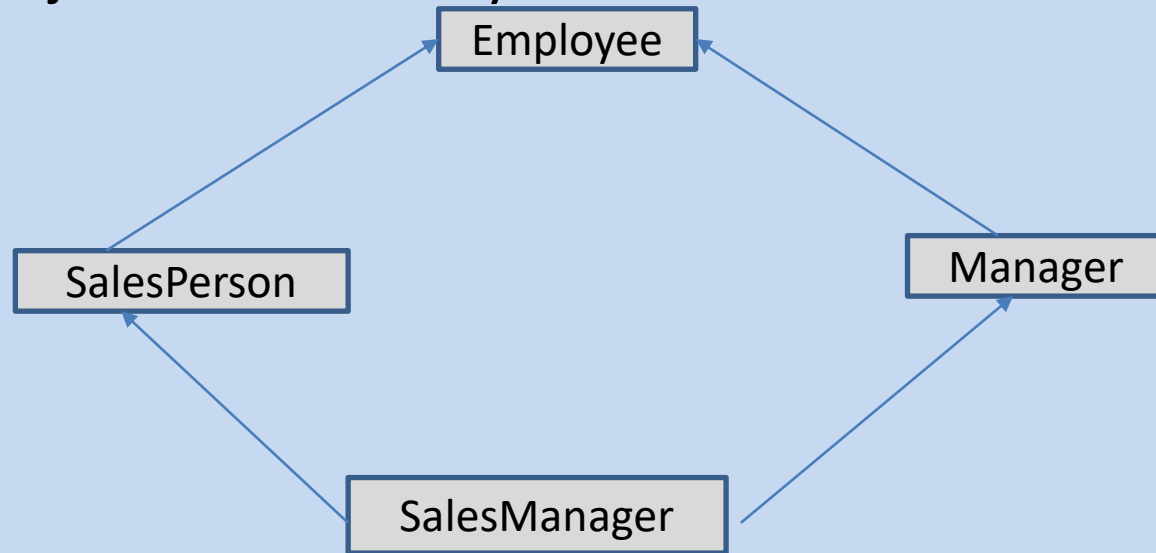


Ambiguities in Diamond Inheritance

- Ambiguity in calling the function.
 - It is similar problem discussed in multiple inheritance.
 - It can be resolved by same way
 - `salObj.Employee::func()`
 - Override func in SalesManager class.
- Data Duplication
 - It happens when the derived class gets multiple copies of the same base class.

Diamond Inheritance

- So if we consider below example and Create an object of SalesManager class –then
- Object will have two copies of id and name from ,which increases size of object unnecessarily.



Virtual Base class

- Duplicate data member ambiguity can be resolved by declaring Virtual base class.
- Derive SalesPerson and Manager class using virtual keyword from Employees
- Then derive Manager class from both of them.

Virtual Base class

- By declaring base class as virtual ,duplicate copies of base class data member is not created..
- There is only one copy of common base class in memory and it's pointer reference is there in derived class object.
- The meaning of virtual keyword is overloaded.

Virtual Base class

```
Class SalesPerson:virtual public Employee
{
    .....
};
```

```
Class Manager:virtual public Manager
{
    .....
};
```

When inheritance is implemented using virtual base class it is termed as Virtual Inheritance.

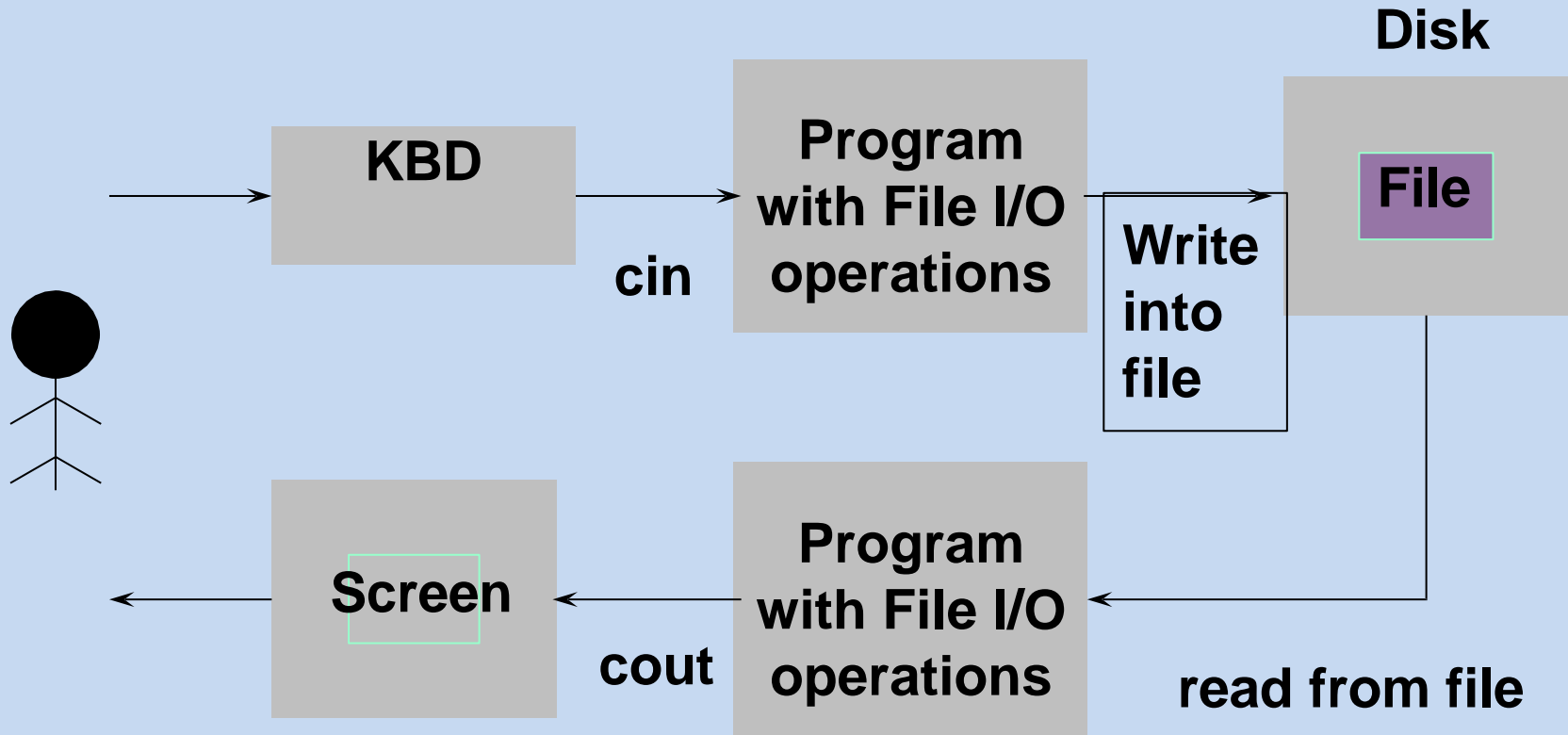
File Handling

File I/O

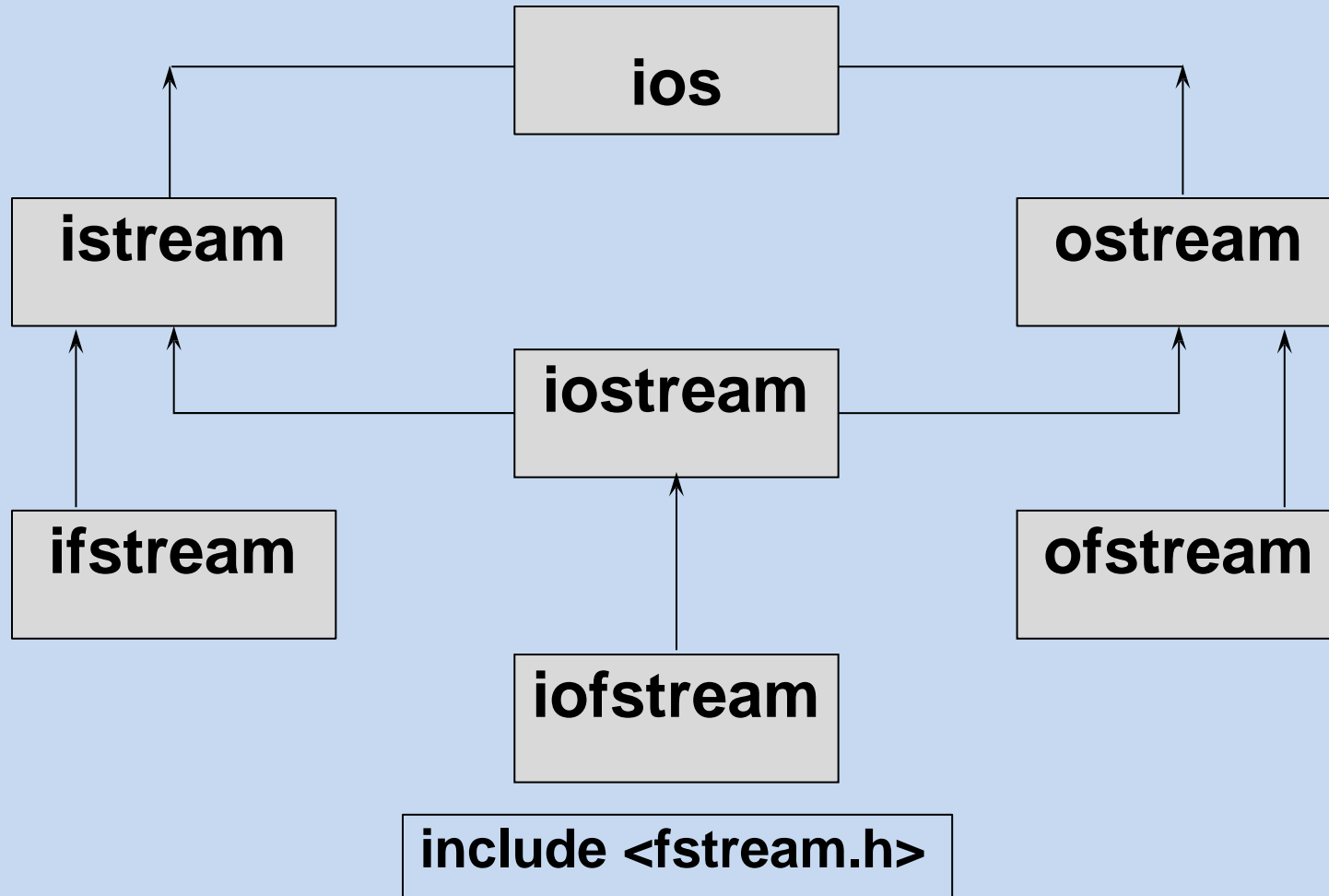
Data which outlives the program is stored on a disk in the form of files

A file is a collection of related data stored on a disk

Data Flow for File I/O



Stream Classes for File I/O



Handling Files

- To read / write data to /from a file
 - Open a file
 - Do read write operations
 - Close file
- File can be opened in two ways
 - Using construction function of a class
 - Using member function `open()` of class

Using constructor to open a file

```
int main() {  
  
    ofstream outfile("result");  
    // outfile is object of class ofstream  
    // constructor will open it in write mode  
  
    char ch = ' ';  
    outfile.put( '1' ).put( ' ' ).put( ch );  
    outfile << "1 + 1 = " << (1 + 1) << endl;  
  
    return 0;  
}
```

👉 inserts in result file 1) $1 + 1 = 2$

When file objects goes out of scope corresponding destructor is called which closes the file

Using open() to Open File

```
Void main {  
  
    ofstream outfile;           // create output stream  
    outfile.open("Result1");    // open a file  
  
    ...  
  
    outfile.close();            // to disconnect stream  
  
    outfile.open("Result2");  
  
    ...  
  
    outfile.close();  
  
}
```

To handle multiple files simultaneously create more streams

e.g. ifstream fin1, fin2;

More About open() : mode

Parameter	Meaning
<code>ios::app</code>	Append to an end of file
<code>ios::binary</code>	Binary file
<code>ios::in</code>	Open file for reading only (default arg)
<code>ios::nocreate</code>	Open fails if the file does not exist
<code>ios::noreplace</code>	Open fails if the file already exists
<code>ios::out</code>	Open file for writing only (default arg)
■ The mode can combine two or more parameters using bitwise OR	
■ e.g.: <code>fout.open("data", ios::app ios::nocreate)</code>	

Handling Files

- A pair of function `put()` and `get()` is used to access the file character by character
- A `getline()` is used to read a file line by line
- A pair of functions `write()` and `read()` are used to access the file in binary mode
- Objects with formatted i/os are handled in a binary file using `read()` / `write()` functions
- An `ifstream` object, such as `fin`, returns a value of 0 if error occurs in the file operation including the end-of-file condition

Need of Dynamic Cast

```
Class A
{
    public:
    virtual void show()=0;
};
Class B :public A
{
    public:
    Void show()
    {
        .....
    }
    Void display()
    {
        .....
    }
}
```

```
Int main()
```

```
{
```

```
    A *Aptr=new B;
```

```
    Aptr->show();
```

```
    Aptr->display();
```

```
}
```

Aptr->display();---Gives Compile time error.as display() is specific to class B.

So what is the Solution?

Dynamic Cast

- Performs type conversion at run time.
 - Type-safe down casting
 - Guarantees the conversion of base class pointer to as derived class pointer.
- Works with pointer only and not with objects.

Dynamic Cast

```
int main()
{
    A *Aptr=new B();
    Aptr->show();
    B *bptr=dynamic_cast<B*>(Aptr);
    If(bptr)
        bptr->display();

    return 0;
}
```

Reinterpret_Cast

- Performs low level interpretation of bit pattern. It allows to edit individual bytes in memory by using bitwise/bitshift operator.
- Used to convert any data type to any other data type.
- It should not be used to cast down a class hierarchy or to remove the const or volatiles qualifiers.

```
Complex <double>*com;  
Char *pc=reinterpret_cast<char*>(com);
```

Need of Namespace

- Software development is team effort.
- It is difficult to control names of variables, structures, classes, functions etc.
- Same variable names, structure names, class names lead towards re-declaration error.
- Same is the case when two or more functions with same signature are in the scope.
- The Solution is –Namespace.

Namespace

Namespace is a work area or a declarative region that attaches an additional identifier to all names declared inside it.

```
namespace space1
```

```
{
```

```
    void f();
```

```
    class A
```

```
    {
```

```
        ....
```

```
    }
```

```
}
```

```
namespace space2
```

```
{
```

```
    class B
```

```
    {
```

```
        ....
```

```
    };
```

```
    void f();
```

```
};
```

Using directive

➤ Using is resolved by compiler as unique

```
int main()  
{  
    using namespace space1;  
    .....  
    using namespace spac2;  
}
```


END

THANK YOU