C++

OOP AND MAJOR PILLARS

Different roles

End User : Who is not a programmer. Who executes your programs. Who is non-technical.

Programmer User : Who writes main( ) and does programming for interaction with end user, who uses class library, call functions, create menus, thinks about end user

Developer User : Who designs classes, functionalities with classes and thinks about programmer user. He is senior and experienced.

Procedural Oriented Programming(POP)	Object Oriented Programming(OOP)
Problem is divided in to small logically independent tasks called as procedures or functions.	Problem is divided in to small real time entities called as objects
Importance is given to different functions and the sequence in which these functions are called	Importance is given to data and not to functions, actually functions are called on real time entities i.e. objects
We use only global and local scope. Does not have any access specifiers as such.	There are 3 access specifiers in C++ public, private and protected
Data is not hidden.	Data may be kept hidden by making it private
Examples : C, COBOL, PASCAL, FORTRAN etc.	Examples: C++, Java, C#, SmallTalk etc.

class

1. Class is classification of an entity
2. It is user-defined data type
3. It is collection of data members and member functions
4. It is blue-print for an object.
5. It is template for an object.
6. We declare class once in our program and create many objects of that class
7. Class is best example of encapsulation
8. At the time defining a class we do data level abstraction.
9. It is a logical entity. (concept)

object

1. It is a variable of type class
2. It is representation of an entity which has state, behavior and identity
3. object's life cycle - object is born, it undergoes many changes, it dies.
4. It is physical entity. (memory is allocated for object when created and memory is released when object goes out of scope).
5. We can create multiple objects of a class. Each object will have its own life-cycle.
6. Value of member variable gives us state of object at any given point of time.
7. We call functions of class with object and state of object is available in function.

Abstraction

Abstraction means showing only essential information while hiding the implementation details.

For example, a person driving a car. The driver knows that pressing the accelerator increases the car’s speed, and applying brakes stops it. However, the driver doesn’t need to understand the complicated inner workings of the car’s mechanisms.

Encapsulation

To keep data and functionalities together in one single unit.

Encapsulation is achieved through private and public access specifiers.

Data members (variables) are kept private and member functions are kept public.

my_laptop.on() - When the function / method is called on a particular object, the state of that object is going to change.

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users.

setter and getter methods

These methods are written for all possible member variables.

setter methods are used to set values of member variables.

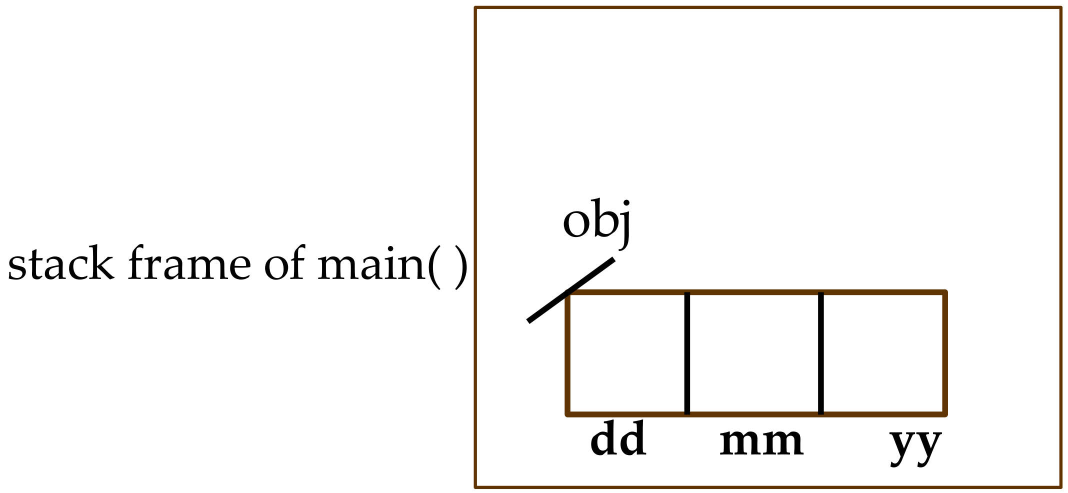
getter methods are used to get values of member variables.

Sample code to understand and write class **date**

```
#include<iostream>
using namespace std;

class date {
   int dd, mm, yy;

public:
   void set_date(int d, int m, int  y)
    {
        dd = d;
        mm = m;
        yy = y;
    }
   void display( )
    {
        cout << “\n”<< dd << “/” << mm <<“ / “<<yy;
    }
};  //end of class
```

```
int main( )
{
     date obj;
     obj.set_date(20,10,2023);
     obj.display( );
     return 0;
}
```

object allocation on stack in RAM

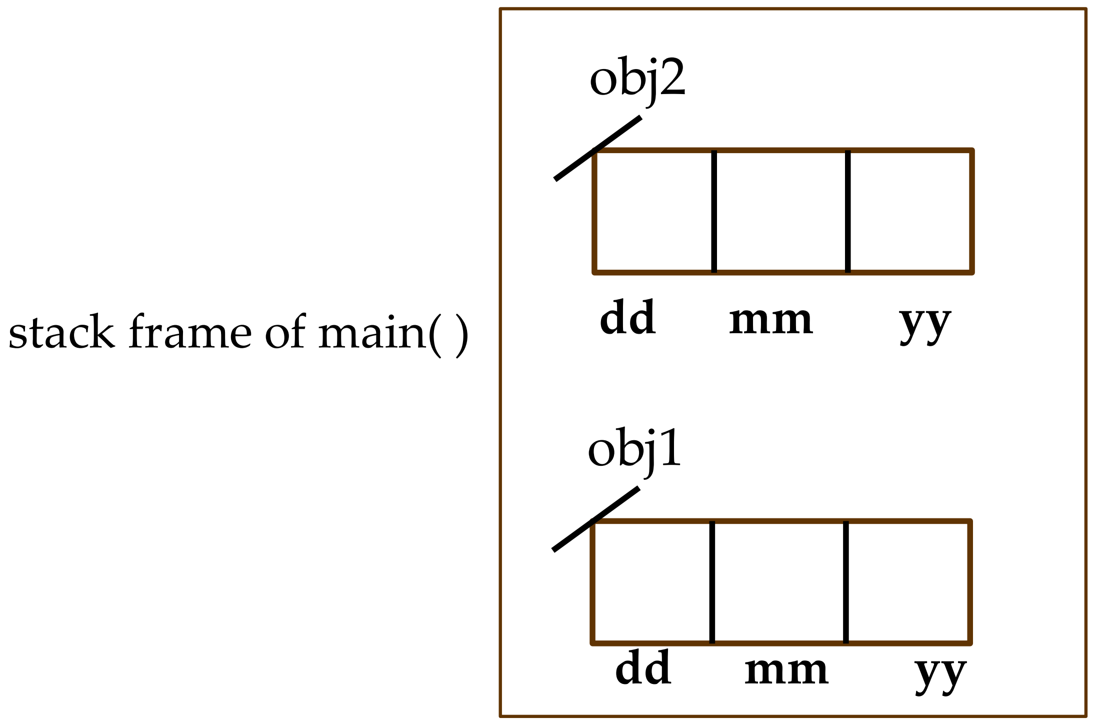


Sample code to understand and write class **date**

```
#include<iostream>
using namespace std;

class date {
   int dd, mm, yy;

public:
   void set_date(int d, int m, int  y)
    {
        dd = d;
        mm = m;
        yy = y;
    }
   void display( )
    {
        cout << “\n”<< dd << “/” << mm <<“ / “<<yy;
    }
};  //end of class
```

```
int main( )
{
     date obj1, obj2;
     obj1.set_date(20,10,2023);
     obj1.display( );
     obj2.set_date(23,10,2023);
     obj2.display( );
     return 0;
}
```

object allocation on stack in RAM



Assignment: Do it yourself.

1. Declare and define class time having hh,mm,ss. Write accept_time( ) and display_time( ) in format “ hh : mm : ss “
2. Declare and define class point having x, y. Write set_point(int, int) and display_point( ) in format (x , y)

OOP Terminologies

Members variables / data members / fields: variables declared in the class

Member functions / methods: functions declared in the class

```
class date {
  int dd, mm, yy;                          Member variables

public:
  void set_date(int dd, int mm, int  yy)
   {
       …
       …                                   Member function
   }
  void display( )
   {                                       Member function
   }
};  //end of class
```

We can declare as many member variables and member functions in a class.

this pointer

- **this pointer** is passed implicitly in every function of class and it contains address of calling object.
- Each object gets its own copy of the data member inside member function through this pointer.
- The ‘this’ pointer is passed as a hidden argument to all member function calls and is available as a local variable within the body of all functions.
- Following are the situations where ‘this’ pointer is used:
  1) When local variable’s name is same as member’s name
  2) To return reference of the calling object from function.

Sample code to understand **this Pointers**

```
#include<iostream>
using namespace std;

class date {
   int dd, mm, yy;

public:
   void set_date(int dd, int mm, int  yy)
    {
        this->dd = dd;
        this->mm = mm;
        this->yy = yy;
    }
   void display( )
    {
        cout << “\n”<< dd << “/” << mm <<“ / “<<yy;
    }
};  //end of class
```

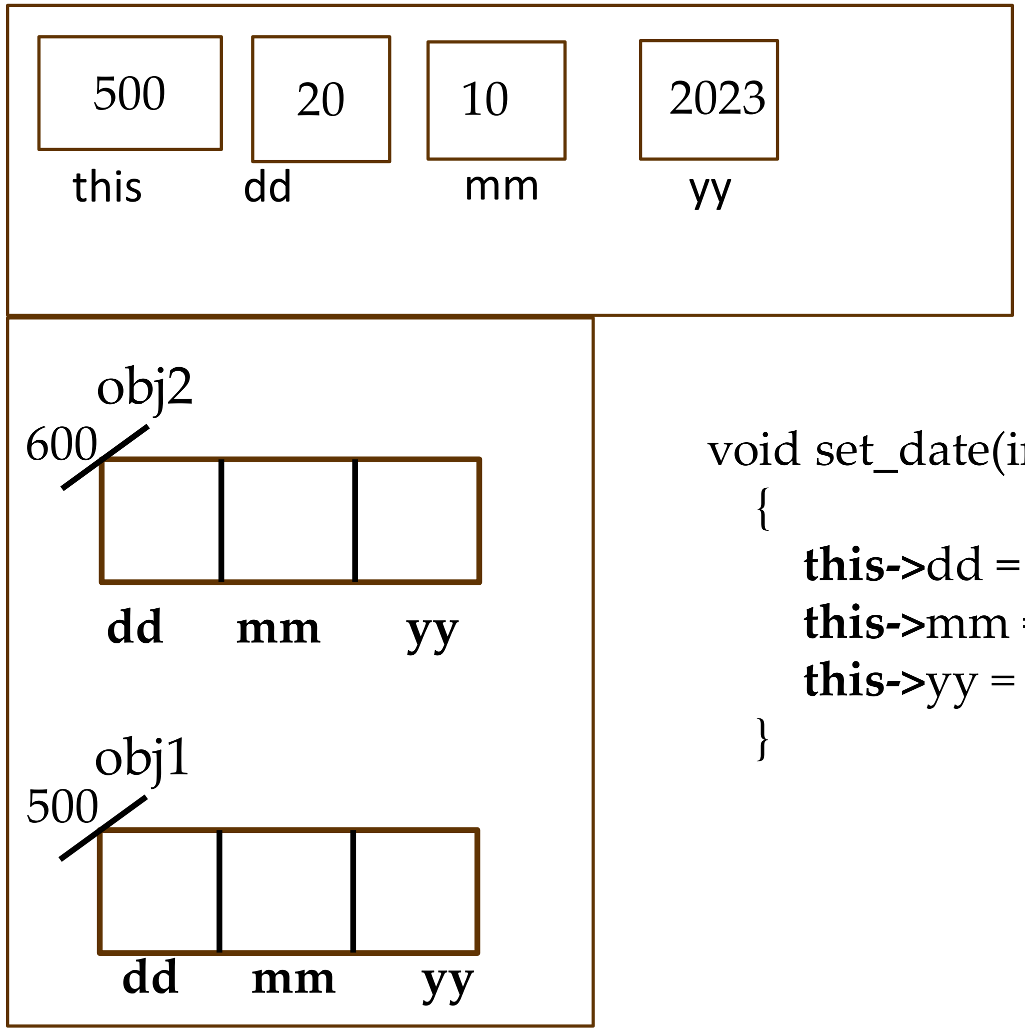
```
int main( )
{
     date obj1, obj2;
     obj1.set_date(20,10,2023);
     obj1.display( );
     obj2.set_date(23,10,2023);
     obj2.display( );
     return 0;
}
```

Stack frames in RAM on function call

obj1.set_date(20,10,2023);

stack frame of set_data( )

stack frame of main( )



```
void set_date(int dd, int mm, int  yy)
  {
      this->dd = dd;
      this->mm = mm;
      this->yy = yy;
  }
```

Access Specifiers in C++

Private :

- We can access the private members from within the class only.
- We can not access the private members from outside the class.
- We keep data members as private, so that user of class can not modify the data from out side the class.
- We can declare some of the functions as private if needed. These functions will not be accessible from outside the class.

Access Specifiers in C++

Public :

- We keep member functions as public, because we want that the user of the class to be able to call functions using object of the class.
- Public means we can access the members from within the class as well from outside class.
- Public is used to provide interface of class. User of the class can talk with the object by calling public functions.

setter and getter methods

Data is kept private in the class. To access or modify the data members we write methods for all possible member variables. These methods are getters and setters. We can implement all possible data validations in these methods.

- **setter methods are used to set values of member variables.**

- **getter methods are used to get values of member variables.**

Setters and getters in date class

```
void setDay(int d)
 {
   if( d <= 31)  dd = d;
   else cout<<"\n Invalid day.. can not be set as value of dd...";
 }

 void setMonth(int m)
 {
   if( m <= 12)  mm = m;
   else cout<<"\n Invalid month.. can not be set as value of mm...";
 }

 void setYear(int y)
 {
   yy = y;
 }
```

```
int getDay()
 {
   return dd;
 }

 int getMonth()
 {
   return mm;
 }

 int getYear()
 {
   return yy;
 }
```

Assignment:

Add getters and setters in **time** and **point** class, and call these methods from main( ).

Constructor

- It is a public member function of a class
- Name of constructor function is same as name of class
- It is called automatically when object of the class created / object is born
- We can overload constructors – default constructor and parameterized constructor
- Constructors are used to initialize values of data members of class and we can allocate resources for objects in constructor. (resources means memory / any device)

- Constructors do not have any return data type, not even void
- If no constructor is written in class definition, compiler provides default constructor.
- If user is writing any constructor, compiler will not provide its default copy.

Destructor

- It is a member function of a class
- Name of destructor is ~followed by name of class
- It is called automatically when object goes out of scope / object dies
- We cannot overload destructor. It is one per class
- We release resource allocated in constructor.
- Destructor do not have any return data type.
- If no destructor is written, compiler will provide default destructor.

Assignment: Add constructor and destructor for date, time and point classes.

Assignment: Declare a class student having

data members : int rno, string name, int mk1, mk2, mk3, total, char grade
member functions: constructor, parameterized constructor, destructor, display, getters and setters, calculate_grade

Operator overloading

Need to overload operator-

- Operator Overloading is a feature of C++ because of which additional meanings can be given to existing operators for user-defined datatypes.
- **operator is keyword in C++ which is used to implement** operator overloading.
- Operator overloading feature makes user defined data type (class) more natural & closer to built in data types.

Rules of Overloading Operator

You cannot create new operators, only can overload existing ones.

precedence or associatively of an operator can not be changed

Meaning of operator should be similar as for built-in data types

ways to overload operators in C++

There are various ways to overload Operators in C++ by implementing any of the following types of functions:

1) Member Function

2) Non-Member Function or **Friend Function**

List of operators that cannot be overloaded

1) Scope Resolution Operator (::)

2) Ternary or Conditional Operator (?:)

3) Member Access or Dot operator (.)

4) Pointer-to-member Operator (.*)

5) Object size Operator (sizeof)

6) Object type Operator(typeid)

7) static_cast (casting operator)

8) const_cast (casting operator)

9) reinterpret_cast (casting operator)

10) dynamic_cast (casting operator)

Copy constructor

- When an object is created using another object, copy constructor is called implicitly.
- Compiler provides default copy constructor in case the programmer is not providing.
- Default constructor does bit by bit copy of one object to another.
- It is called when we are passing object of any class as call by value parameter
- It is also called when any function is returning object
- We can create copy of object at the time of declaration. Syntax is

  date obj2(obj1)

  Here date is the name of class, obj2 is the new object to be created and it will be copy of obj1

Copy constructor

- When an object is created using another object, copy constructor is called implicitly.
- Compiler provides default copy constructor in case the programmer is not providing.
- Default constructor does bit by bit copy of one object to another.
- It is called when we are passing object of any class as call by value parameter
- It is also called when any function is returning object
- We can create copy of object at the time of declaration. Syntax is

  ```
  date obj2(obj1)
  ```

  Here date is the name of class, obj2 is the new object to be created and it will be copy of obj1

```
class myArray{
  int size;
  int *ptr;

  public:

      myArray( ) {
          size = 5;
          ptr = new int[size];
          for(int i=0 ;i<size ; i++)
              ptr[i] = 0;
      }

    ~myArray( ) {
         delete []ptr;
    }
};
```

Scenario 1:

```
int main( ) {
    myArray obj1;
    return 0;
}
```

Scenario 2:

```
int main( ) {
    myArray obj1;
    myArray obj2;
    return 0;
}
```

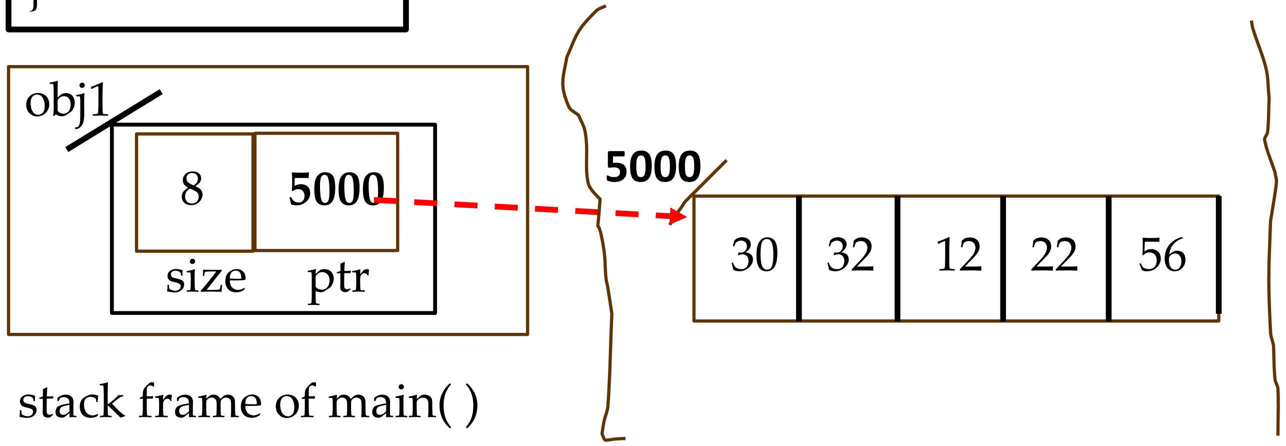
Scenario 3:

```
int main( ) {
    myArray obj1;
    myArray obj2(obj1);
    return 0;
}
```

```
class myArray{
 int size;
 int *ptr;

 public:

    myArray( ) {
        size = 5;
        ptr = new int[size];
        for(int i=0 ;i<size ; i++)
            ptr[i] = 0;
    }

   ~myArray( ) {
        delete []ptr;
   }
};
```

Scenario 1:

```
int main( ) {
    myArray obj1;
    return 0;
}
```



stack frame of main( )

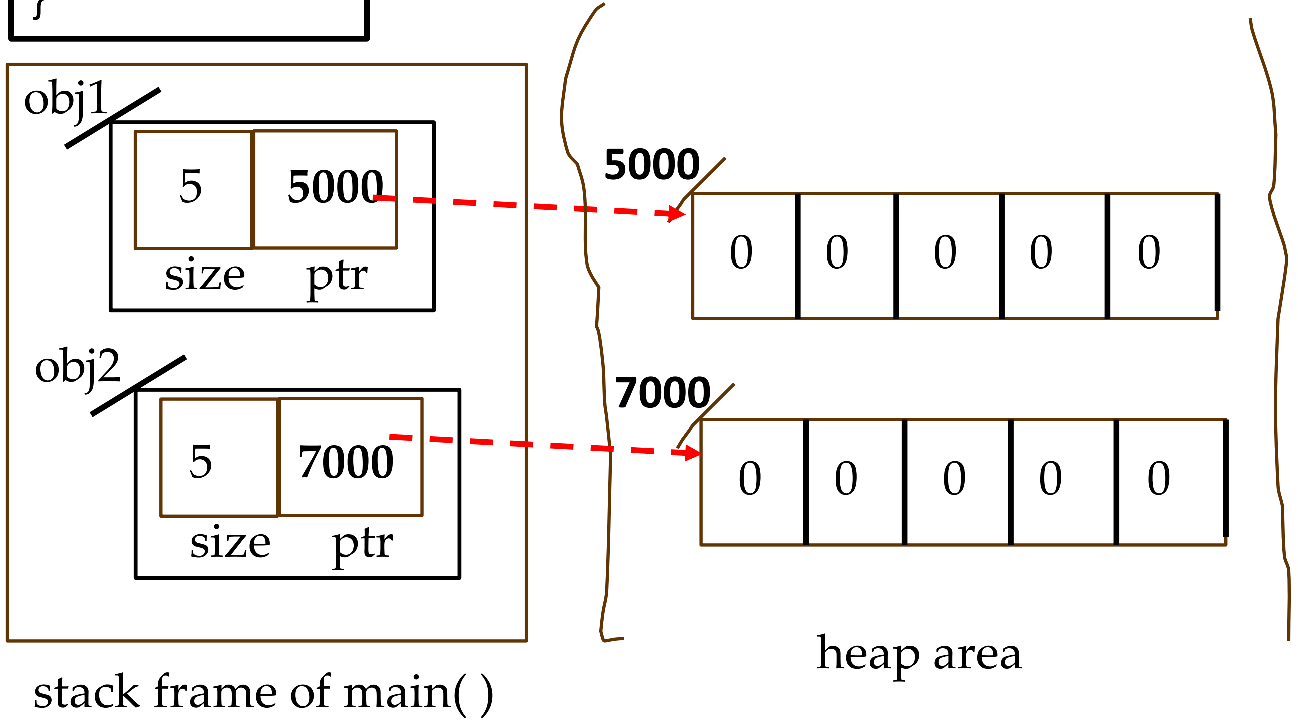
heap area

heap area

```
class myArray{
  int size;
  int *ptr;

  public:

     myArray( ) {
        size = 5;
        ptr = new int[size];
        for(int i=0 ;i<size ; i++)
            ptr[i] = 0;
     }

   ~myArray( ) {
       delete []ptr;
   }
};
```

Scenario 2:

```
int main( ) {
    myArray obj1;
    myArray obj2;
    return 0;
}
```



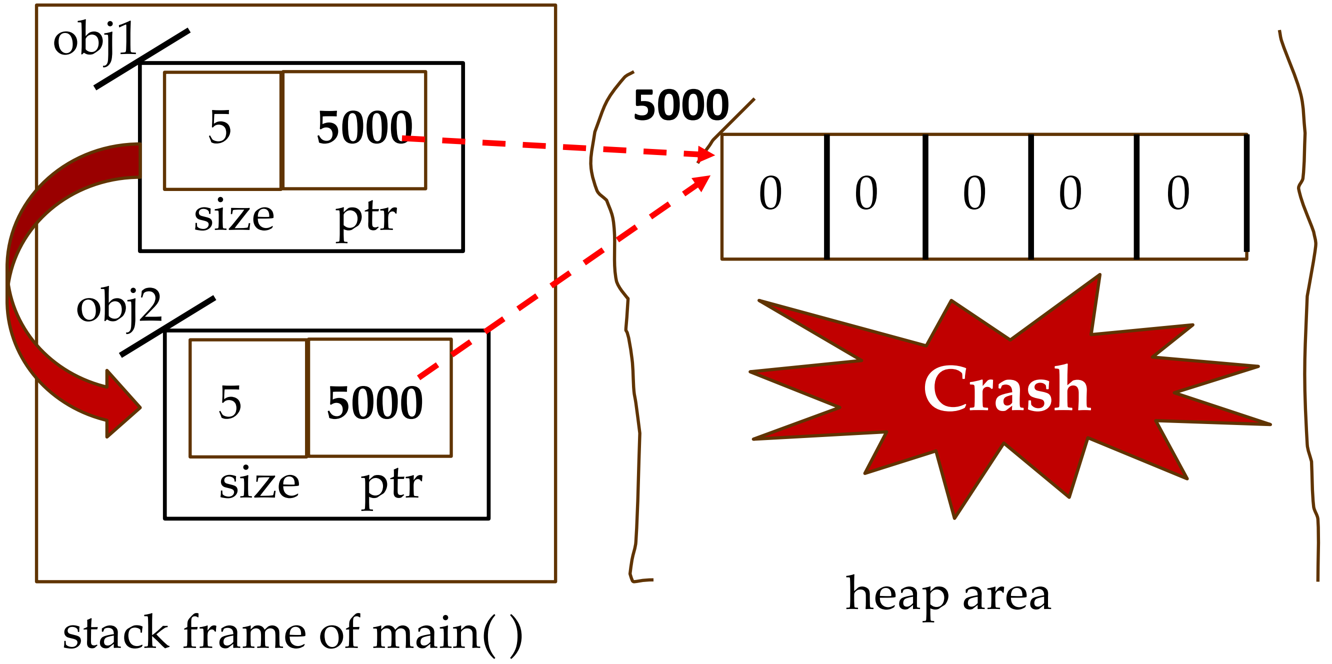
heap area

Shallow copy
Bit by bit copy

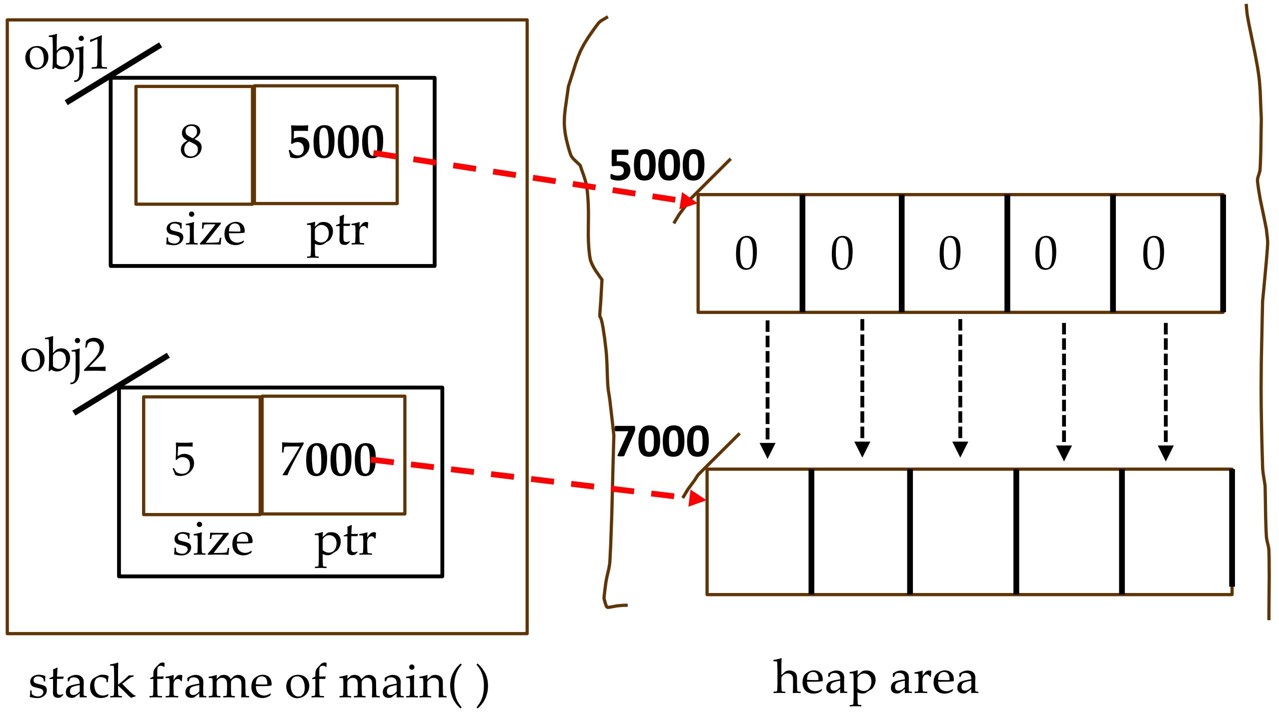
```
class myArray{
 int size;
 int *ptr;

 public:

    myArray( ) {
        size = 5;
        ptr = new int[size];
        for(int i=0 ;i<size ; i++)
            ptr[i] = 0;
    }

  ~myArray( ) {
      delete []ptr;
   }
};
```

Scenario 3:

```
int main( ) {
   myArray obj1;
   myArray obj2(obj1);
   return 0;
}
```



Deep copy by copy constructor



```
myArray(myArray &m ) {
        size = m.size;
        ptr = new int[size];
        for(int i=0 ;i<size ; i++)
            ptr[i] = m.ptr[i];
    }
```

- *Copy constructor has to copy data on stack from source object to target object*
- *Copy constructor has to allocate new memory for pointer variables in target object, then copy contents on heap area from source object to target object.*

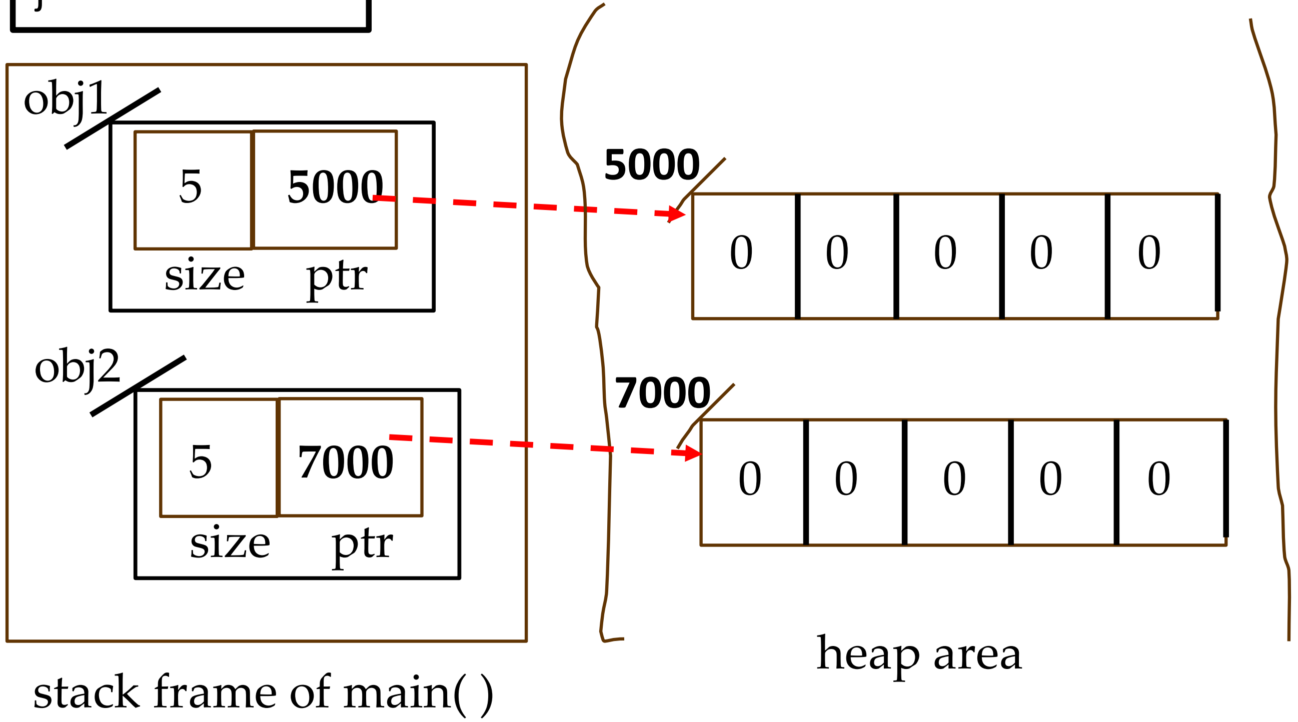
Assignment operator

- When an object is assigned value of another object, assignment operator functions is called implicitly.
- Compiler provides default assignment operator implementation (=) in case the programmer is not providing.
- Default assignment operator function does bit by bit copy of one object to another.
- obj2 = obj1, when obj1 and obj2 are already declared objects of any class.

Consider scenario

```
class myArray{
 int size;
 int *ptr;

 public:

     myArray( ) {
         size = 5;
         ptr = new int[size];
         for(int i=0 ;i<size ; i++)
             ptr[i] = 0;
    }

    ~myArray( ) {
        delete []ptr;
    }
};
```

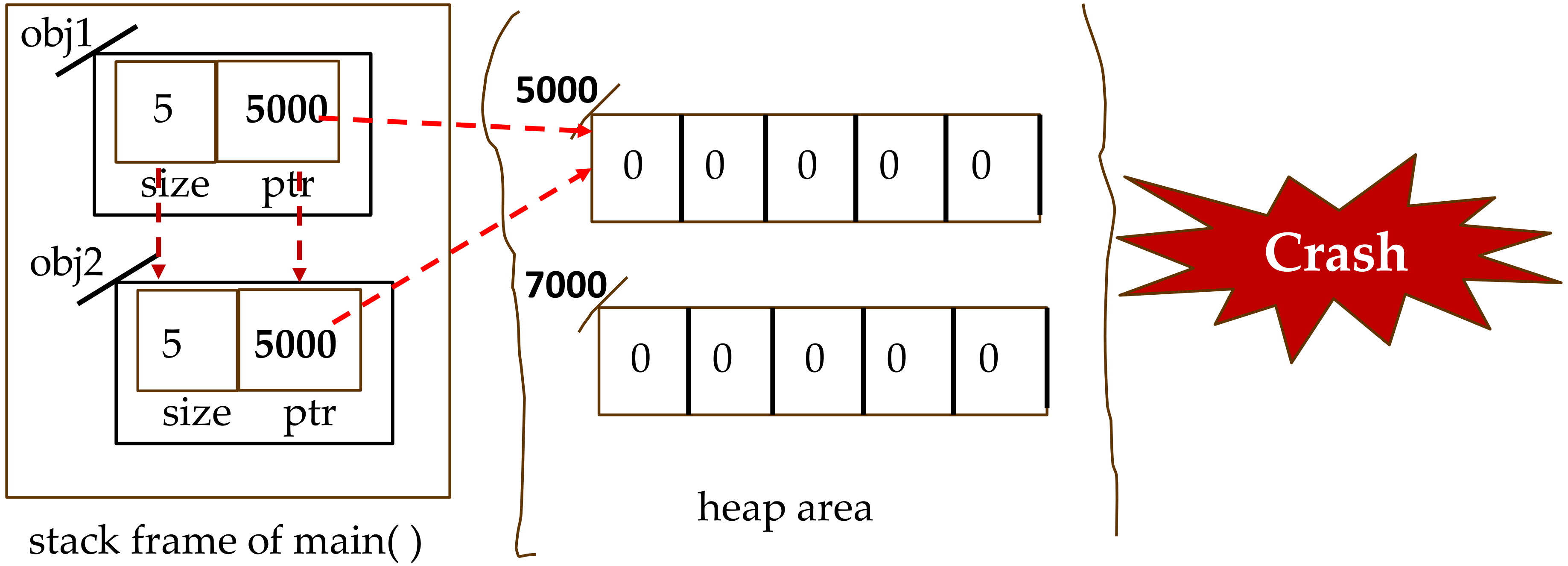
```
int main( ) {
    myArray obj1;
    myArray obj2;
    return 0;
}
```



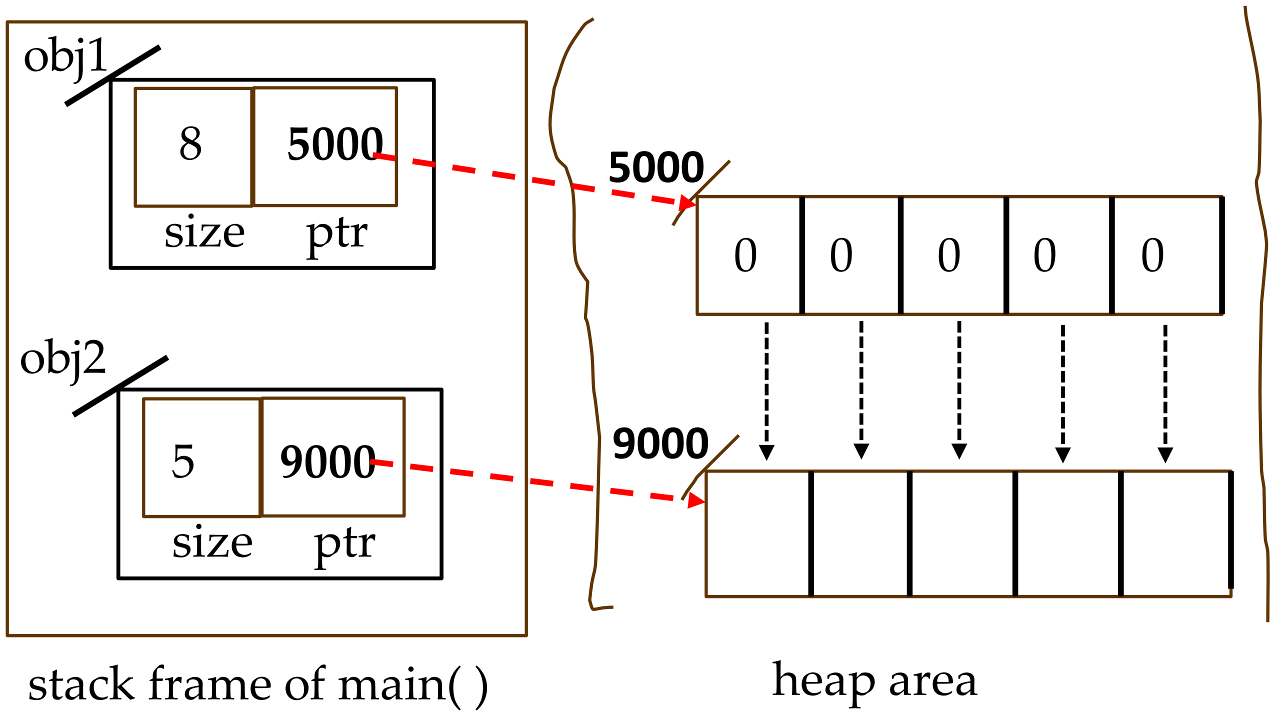
heap area

Default behaviour of assignment operator

```
int main( ) {
   myArray obj1;
   myArray obj2;

   obj2 = obj1; //call to assignment operator
   return 0;
}
```



Assignment Operator



```
myArray operator = (myArray &m ) {
        delete [ ] ptr;
        size = m.size;
        ptr = new int[size];
        for(int i=0 ;i<size ; i++)
            ptr[i] = m.ptr[i];
    }
```

- *Assignment operator has to release memory in hold for ptr of target object.*
- *Assignment operator has to allocate new memory for ptr of target object.*
- *Assignment operator has to copy values of source object to target object.*

Assignment operator

- Some compiler make it compulsory to define signature of assignment operator as

  const &myArray operator = (const myArray &m) { ……. }

  This means the function returns reference to constant and takes constant reference as parameter.

Scope / types of variable

Global variables: As the name suggests, Global Variables can be accessed from any part of the program.

- They are available through out the life time of a program.
- They are declared at the top of the program outside all of the functions or blocks.
- Memory allocation for global variable is on data segment.

Scope / types of variable

Local variables: Variables defined within a function or block are said to be local to those functions.

- Local variables are declared inside a block.
- These variables are created when entered into the block or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access this variable only within that block.
- Initialization of Local Variable is Mandatory.
- Memory allocation for local variable is on function's stack frame on stack segment.

Scope / types of variable

Instance Variables: Instance variables are non-static variables and are declared in a class outside any method, constructor, or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier private default access specifier will be used.
- Initialization of Instance Variable is also Mandatory in C++.
- Instance Variable can be used and accessed only by creating objects.
- Memory allocation is with-in object, when object is created.

Scope / types of variable

Static Variables: Static variables are also known as **Class level variables** and are declared in a class outside any method, constructor, or block.

- Static variables are declared using the **static** keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- Memory allocation is on data segment, scope is limited to class in which it is declared.
- We need to declare static variable outside the class too. Also we can declare global static variable.

Containment

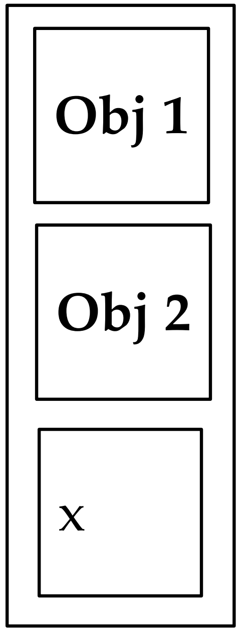
- Containment represents “has a” relationship
- Containment Relationship means the use of an object of a class as a member of another class.

  Ex. Birth_Date or joining date as a part of Employee class
- The container relationship brings reusability of code.

  Ex. Already written Date class can be used in Class Employee.

“HAS-A” Type of Relationship

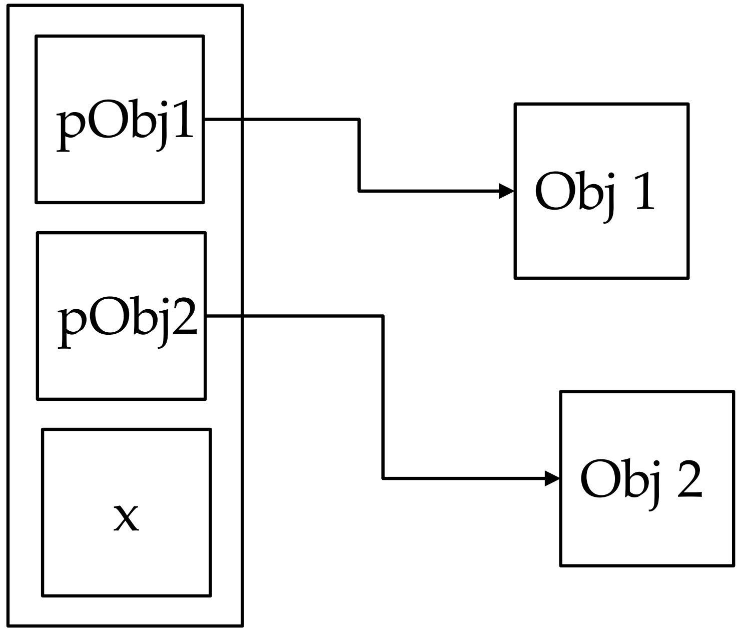
Physical containment



- Also called as tight coupling
- Example

  Car-Engine

Logical containment



- Also called as lose coupling
- Example

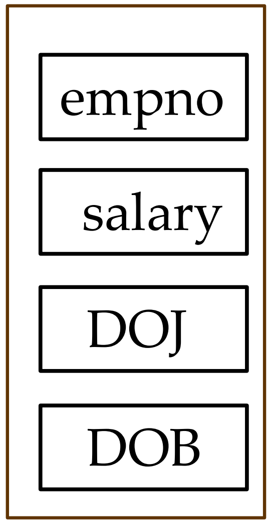
  Car-Documents

Has-a relationship

- Facilitates code reuse
- Already written classes can be used as members of another class
- Example:

Already written class Date can be used as date members of class Person, Employee or Student as DOB, joining_date etc.

```
class employee{
    int empno;
    float salary;
    Date DOJ, DOB;
};
```



Constructor and Destructor

- Contained objects created first
- Order of creation decided by declarations in container class
- Default constructor gets called
- Object destruction takes place in reverse order

Member initialization list

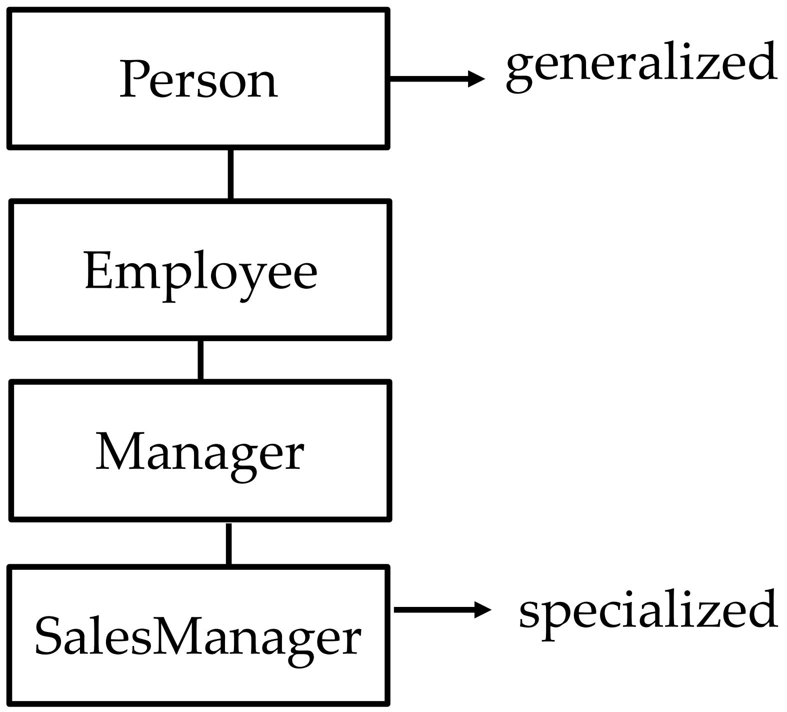
- Container object gets all the data from user and distributes it to appropriate contained object and to itself as well
- If not specified, all contained objects get created using default constructor. User input is reassigned later using member functions
- Call appropriate constructor right at the time of creation. This improves performance
- Even built-in data members can be initialized in this list
- Reference members can be initialized only in this list

Inheritance

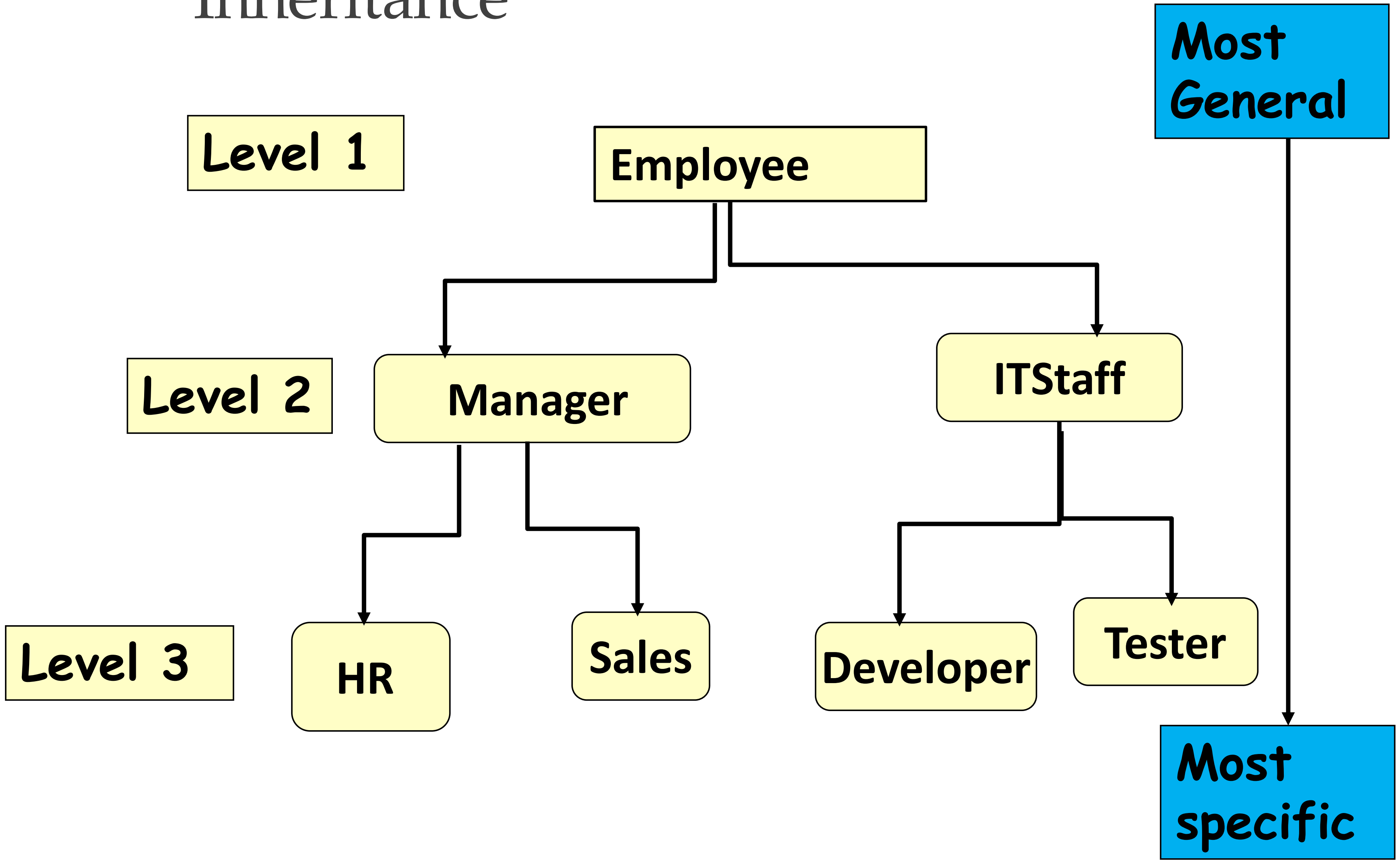
- Inheritance is one of the most powerful feature of object-oriented programming.
- Inheritance is the process of creating new classes, called derived classes, from existing or base classes.
- The derived class inherits all the functionalities of the base class and can add functionalities of its own.
- The base class is unchanged by this process.
- An important result of inheritance is reusability and the ease of distributing class libraries. A programmer can use a class created by another person or company.

Inheritance

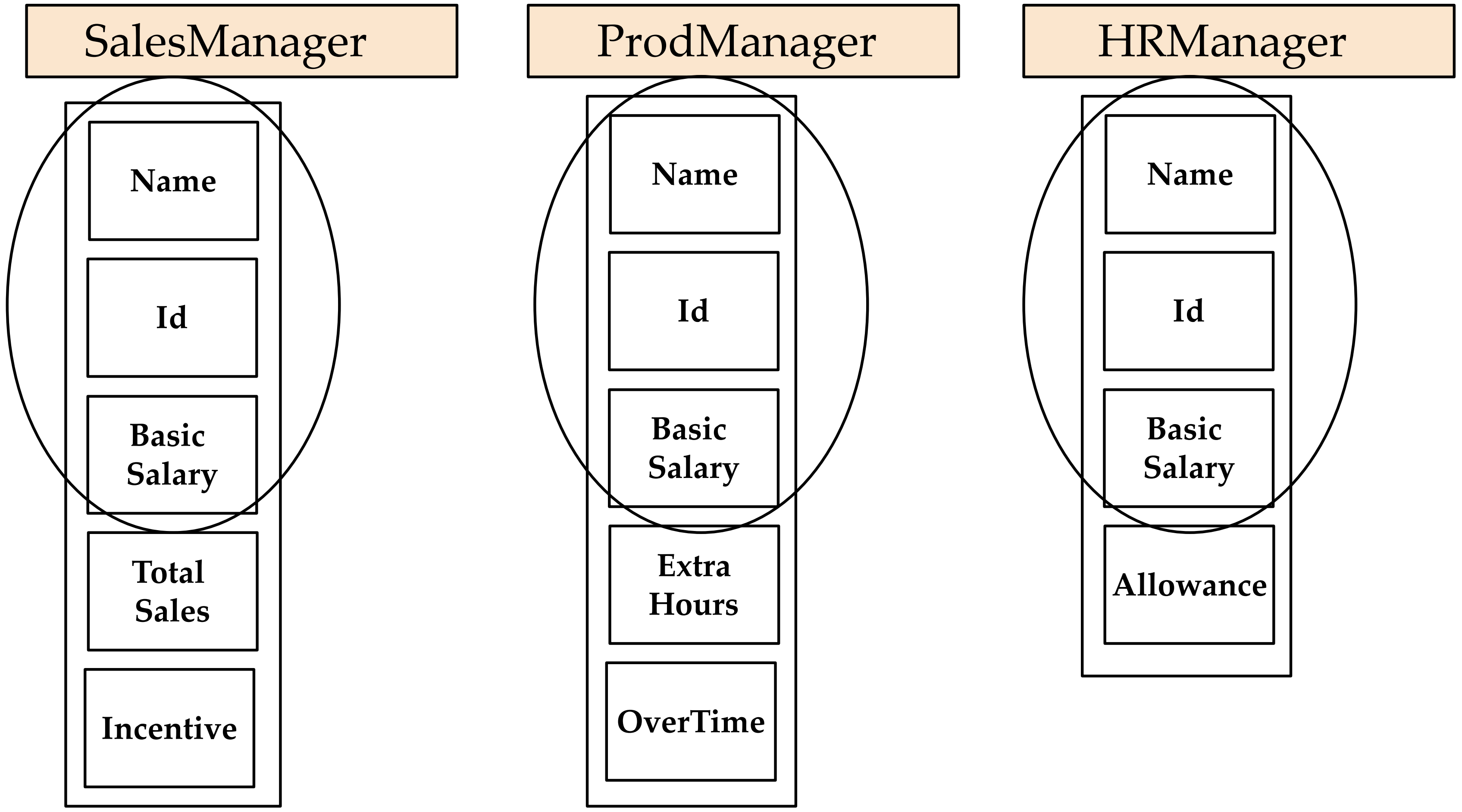
- Establishes ‘is-a’ kind of relationship
- Moves down from generalization to specialization.
- Most general at top to most specific at the bottom of inheritance chain



Inheritance



“IS-A” Type of Relationship



Syntax For Inheritance

Syntax:

class *DerivedClassName* : **access-level** *BaseClassName*

where

- **access-level** specifies the type of derivation
  - private by default, or
  - public
  - protected

Any class can serve as a base class

- Thus a derived class can also be a base class

Access Specifier & Inheritance

Base class member access specifier	**Type of Inheritance**		
	Public inheritance	**Protected Inheritance**	**Private Inheritance**
public	Public in derived class. Can be accessed directly by any non-static member functions, friend functions, and non-member functions.	Protected in derived class. Can be accessed directly by any non-static member functions, friend functions.	Private in derived class. Can be accessed directly by any non-static member functions, friend functions.
Protected	Protected in derived class. Can be accessed directly by any non-static member functions, friend functions.	Protected in derived class. Can be accessed directly by any non-static member functions, friend functions.	Private in derived class. Can be accessed directly by any non-static member functions, friend functions.
private	Hidden in derived class. Can be accessed directly by non-static member functions, friend functions through public or protected member function of base class.	Hidden in derived class. Can be accessed directly by non-static member functions, friend functions through public or protected member function of base class.	Hidden in derived class. Can be accessed directly by non-static member functions, friend functions through public or protected member function of base class.

Polymorphism

- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.

- A real-life example of polymorphism: A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

- Another example is: A shape has different implementation of area function depending on whether the shape is circle or rectangle or triangle etc. Here area() function is called on shape variable according to the what type of shape it is.

Polymorphism is achieved using -

Function binding

Generic pointers and Virtual keyword

V-table and V-pointer

Pure virtual function

Abstract class

Virtual Destructor

Function binding: Process of deciding which function to be called at the of execution of code.

▪ Two types of function binding

1. Compile time binding: Static binding
2. Run-time binding: Dynamic binding

▪ Static binding happens when all information needed to call a function is available at the compile-time.

▪ Dynamic binding happens when all information needed for a function call cannot be determined at compile-time.

Static binding means Compile time binding

▪ Compiler decides, at the time of compilation, which function to be called at the point of invocation.

▪ Compiler decisions are based on types of objects or variables only

▪ Example:

Function overloading: Best match on the basis of types of arguments
Invocations using objects: Based on type of object
Invocations using pointers: Based on type of pointer
Invocations using references: Based on type of reference

Dynamic binding means Run-time binding

- Compiler does NOT decide the actual function to be called at the point of invocation.

- Compiler simply substitutes a code at the point of invocation. This code when executed decides the actual function to be called

- Such calls are called as polymorphic function calls

Polymorphic function call :

▪ A call is polymorphic only if it satisfies following two conditions

1. Functions should be declared as virtual

2. Call should be made using either a generic pointer or a generic reference

▪ Polymorphic calls are not resolved by compiler. It simply substitutes a code at the point of invocation

▪ This code when executed, decides the actual function to be called.

▪ The actual function to be called is decided by the type of object pointed by the pointer and NOT by the type of pointer itself. The type of object is decided at run-time. This is polymorphism

Note that…

- Polymorphism is expensive because memory requirement for vTable per class and vPtr per object.

- It also may reduce the performance, but is very negligible.

- Constructors cant be virtual, because the vPtr gets initialized at the time of construction.

- Virtual function should be non static member function of the base class.

- Virtual functions cannot be used as a friend function.

- If a function is declared as virtual in the base class then ,it will be taken as virtual in the derived class even if the keyword virtual is not used

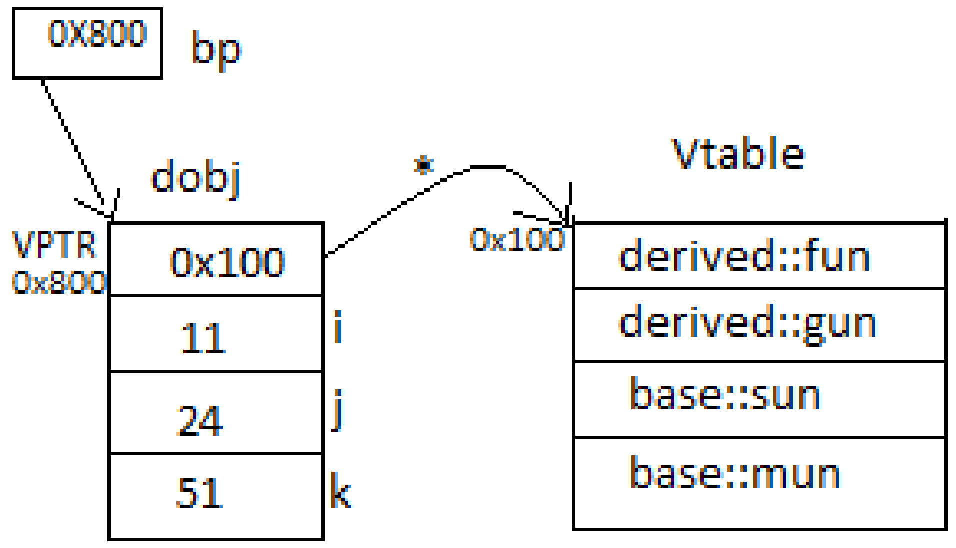
- Polymorphism can also be achieved using references.

Pure virtual function, Abstract Class

- Virtual function without definition is pure virtual function
- Class containing at least one pure virtual function is abstract class
- Abstract class cant be instantiated, its pointers and references can be created
- Class containing only pure virtual functions is pure abstract class, also called as interface
- Derived classes must override pure virtual functions else they are also abstract

Virtual Destructor

- Needed for base class pointer pointing to a dynamically allocated derived class object
- Results into resource leak of derived class if not made virtual
- Use only if needed, else it results into memory overheads in terms of vTable and vPt


0X800
bp
dobj
VPTR
0x800
0x100
11
i
24
j
51
k
*
Vtable
0x100
derived::fun
derived::gun
base::sun
base::mun

Scope Resolution Operator (::)

In C++, the scope resolution operator is ::. It is used for the following purposes.

1) To access a global variable when there is a local variable with same name:

2) To define a function outside a class.

3) To access a class's static variables.

4) In case of multiple Inheritance:

5) Refer to a class inside another class: