We are Not Alternate of VU recorded Lectures we are add on for "LEARNING HUNGERS“ Students so that they can apply VU provided concepts in practical as per software industry need

Object Oriented Programming (C++, Java and C#) (CS304)

Objectives

- Polymorphism

Problem Statement

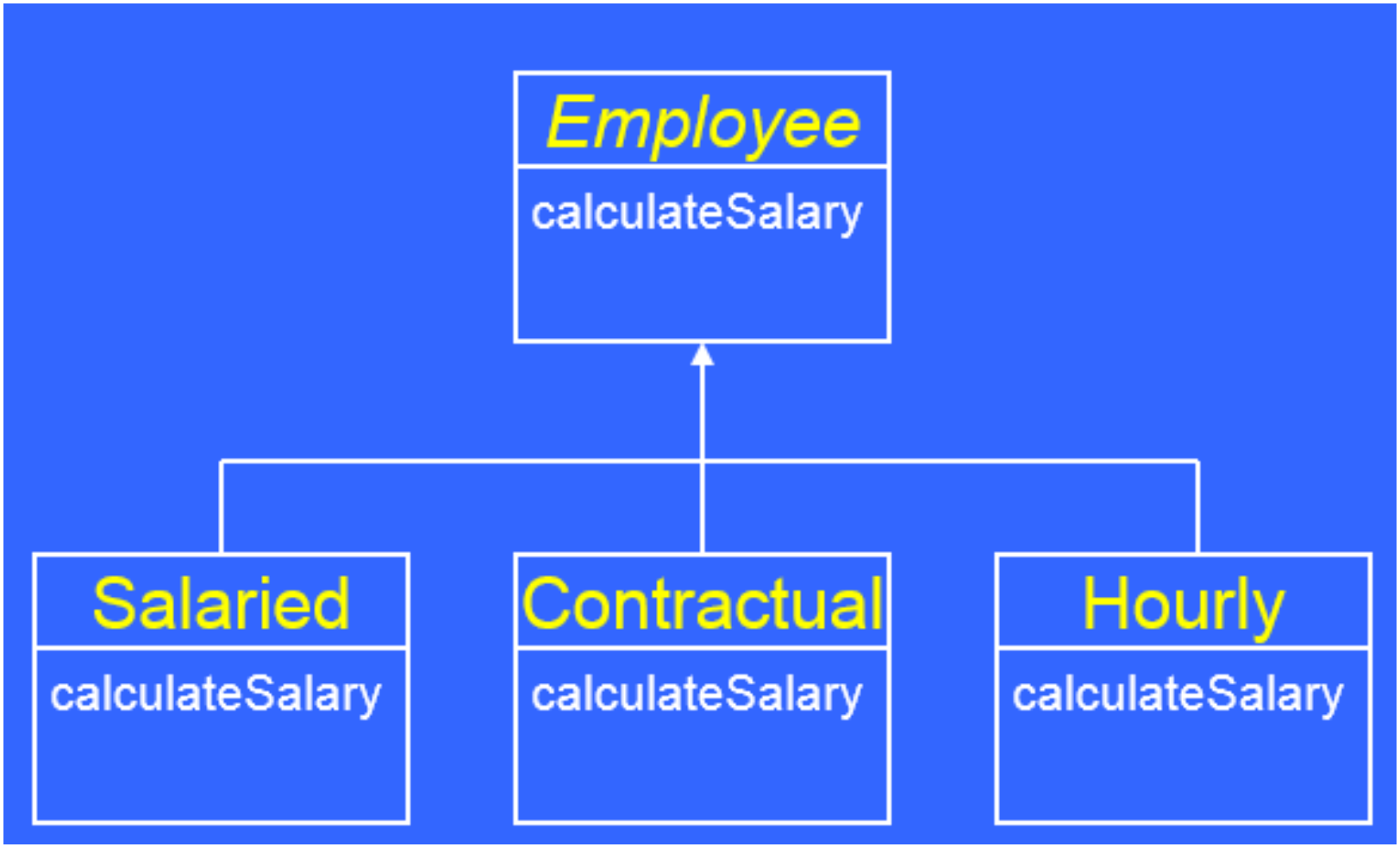
Develop a function that can calculate salary of different Types of Employee based on below Condition

SalariedEmployee = salary + houseRent

ContractualEmployee = salary

HourlyEmployee = hourlyRate * hourWorked

Employee Hierarchy



Employee Hierarchy…..

```
class Employee{
    protected:
       int employeeId;
       string employeeName;
       float salary;
   public :
    .
    .
   void calculateSalary(){
   cout<<"Employee Salary is "<<salary;
  }
};
```

Employee Hierarchy…..

```
class SalariedEmployee : public Employee{
 private:
   float houseRent;
    public:
 void calculateSalary(){
   cout<<"Salaried Employee Salary is
"<<(salary+houseRent);
 }
};
```

Employee Hierarchy…..

```
class ContractualEmployee: public Employee{
 private:
  int contractDuration;
  public :
   void calculateSalary(){

   cout<<"Contract Employee Salary is
"<<salary;
 }

};
```

Employee Hierarchy…..

```
class HourlyEmployee : public Employee{
    protected:
       float hourlyRate;
       float hourWorked;
    public :
  void calculateSalary(){
   salary = hourlyRate * hourWorked;
   cout<<"Hourly Employee Salary is "<<
salary ;
  }

};
```

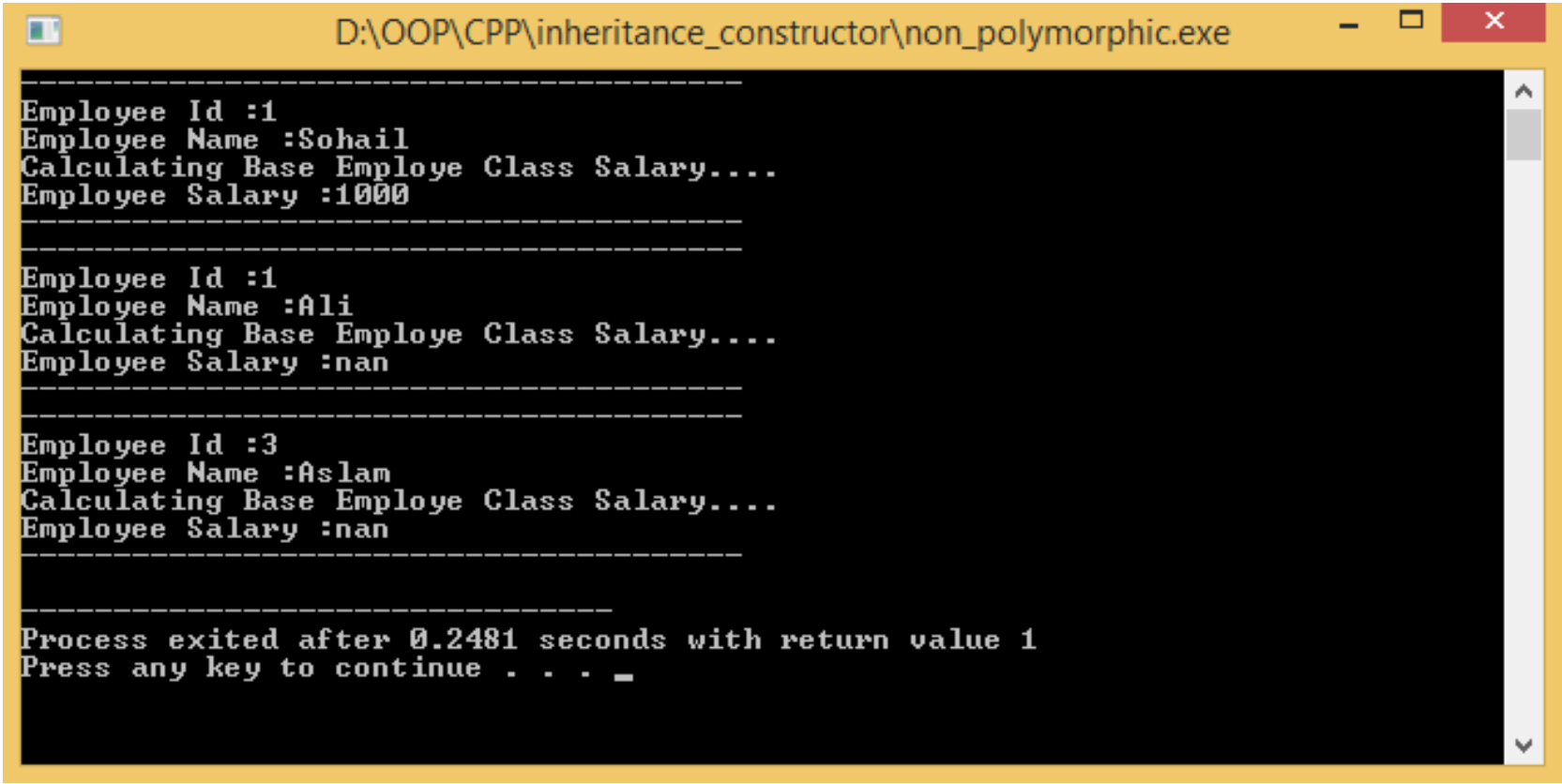
Print Salaries

```
int main() {
      Employee* _emp[ 3 ];
    SalariedEmployee *e1 = new    SalariedEmployee;
    e1->setEmployeeId(1);
    e1->setEmployeeName("Sohail");
    e1->setHouseRent(500.0);
    e1->setSalary(1000.0);
    _emp[0] = e1;
  …
  void printSalaries( _emp, 3 );
  return 0;
}
```

Function printSalary

```
void printSalaries(Employee* _emp[], int size) {
  for (int i = 0; i < size; i++) {
    cout<<"----------------------------------------"<<endl;
    cout<<"Employee Id :"<<_emp[i]->getEmployeeId()<<endl;
    cout<<"Employee Name :"<<_emp[i]->getEmployeeName()<<endl;
    float salary = _emp[i]->calculateSalary();
    cout<<"Employee Salary :"<<salary<<endl;
    cout<<"----------------------------------------"<<endl;
  }
}
```

Output



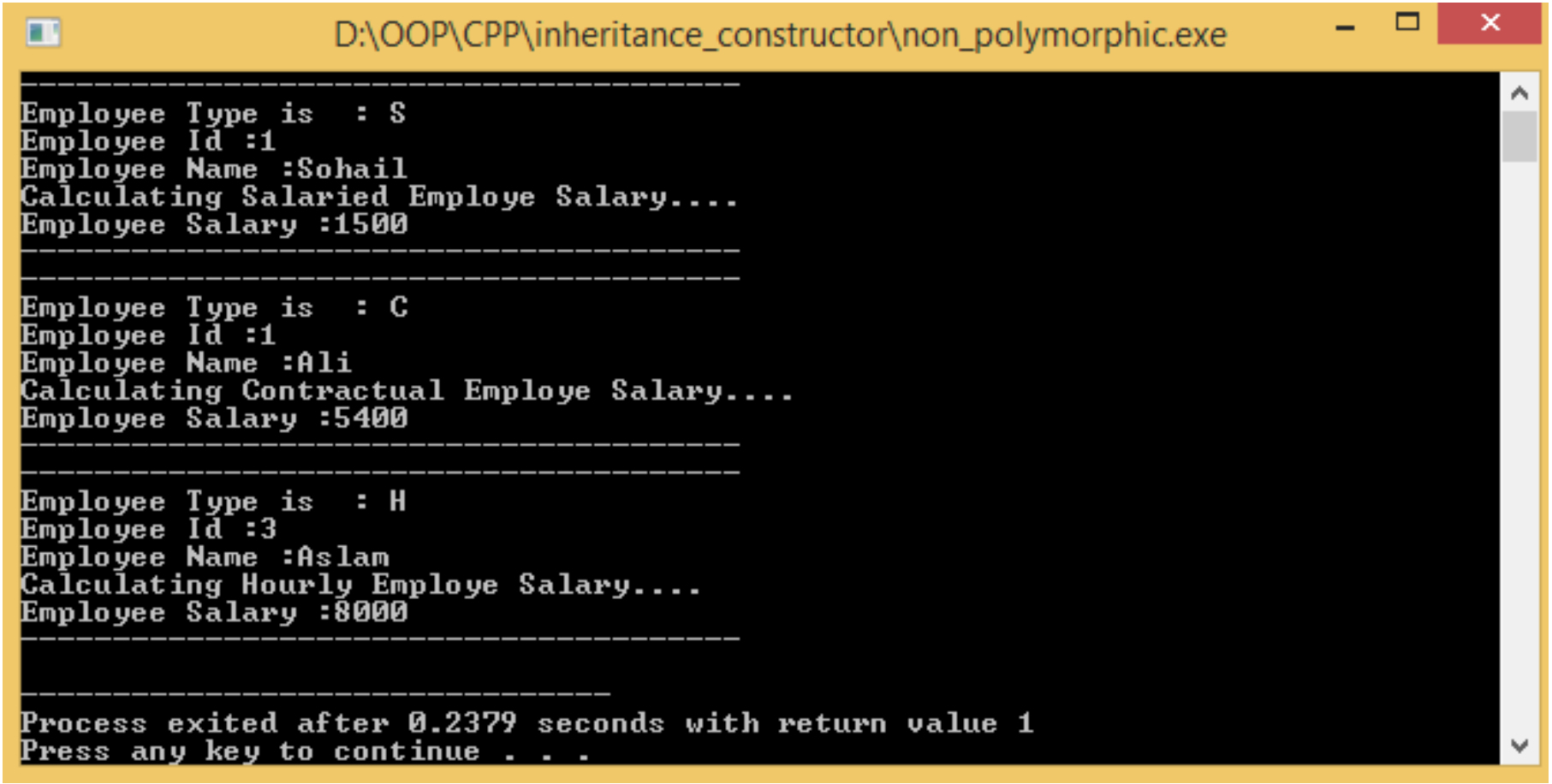
Function printSalary

```
void printSalaries(Employee* _emp[], int size) {
  for (int i = 0; i < size; i++) {
    cout<<"---------------------------------------"<<endl;
    cout<<"Employee Type is  : "<<_emp[i]->getEmployeeType()<<endl;
    cout<<"Employee Id :"<<_emp[i]->getEmployeeId()<<endl;
    cout<<"Employee Name :"<<_emp[i]->getEmployeeName()<<endl;
    float salary = 0.0;

    if(_emp[i]->getEmployeeType()=='C'){
       salary = static_cast<ContractualEmployee*>(_emp[i])-
>calculateSalary();
    }else if(_emp[i]->getEmployeeType()=='S'){
       salary = static_cast<SalariedEmployee*>(_emp[i])-
>calculateSalary();
    }else if(_emp[i]->getEmployeeType()=='H'){
       salary = static_cast<HourlyEmployee*>(_emp[i])->calculateSalary();
    }

    cout<<"Employee Salary :"<<salary<<endl;
    cout<<"---------------------------------------"<<endl;
  }
}
```

Output



Problems ?

- Programmer may forget a check
- May forget to test all the possible cases
- Hard to maintain

Solution?

- To avoid switch, we need a mechanism that can select the message target automatically!

Polymorphism Revisited

- In OO model, polymorphism means that different objects can behave in different ways for the same message (stimulus)
- Consequently, sender of a message does not need to know the exact class of receiver

Virtual Functions

Target of a virtual function call is determined at run-time

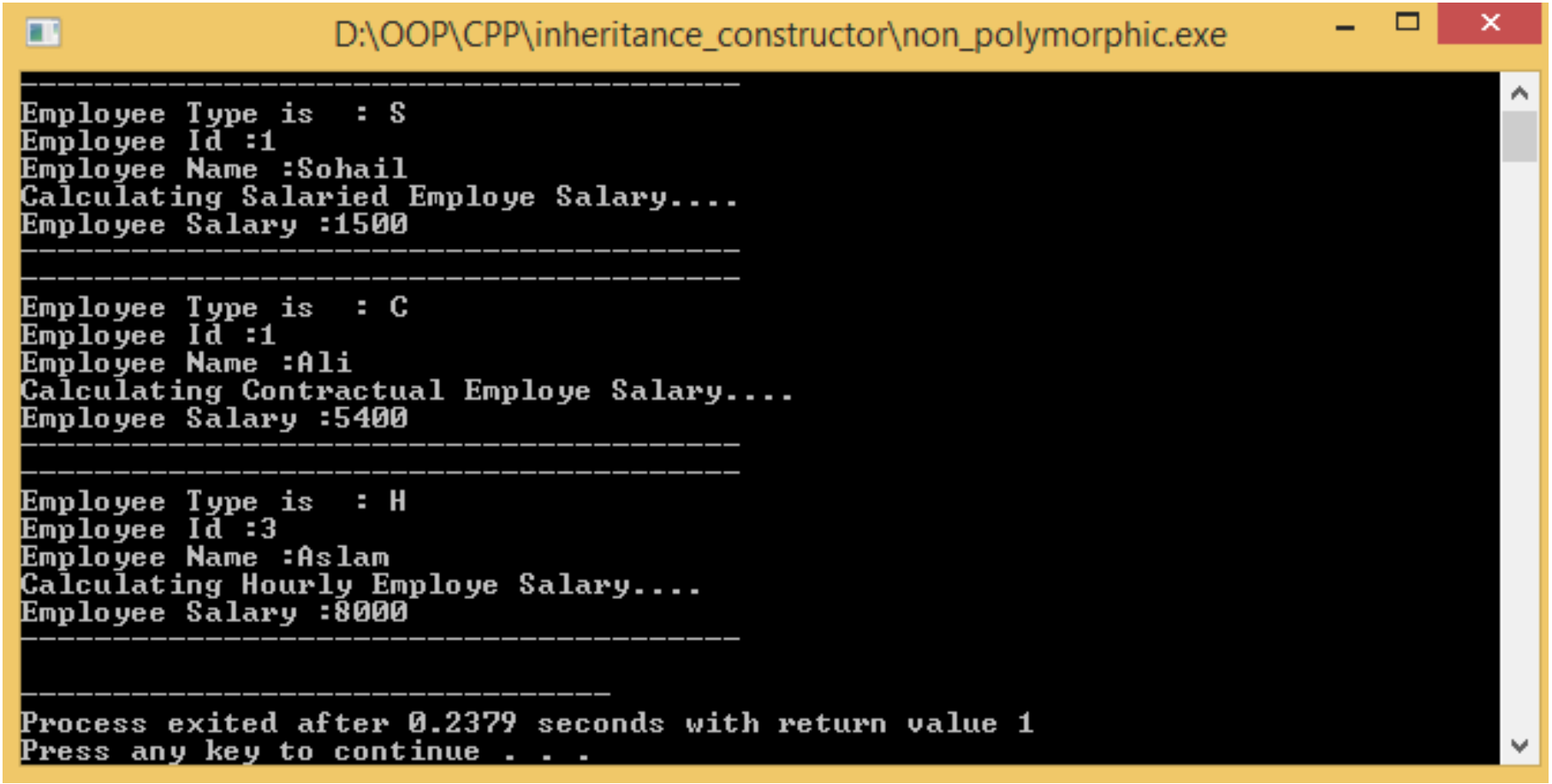
In C++, we declare a function virtual by preceding the function header with keyword “virtual”

```
class Employee {
 …
 virtual float calculateSalary();
}
```

Function printSalary

```
void printSalaries(Employee* _emp[], int size) {
  for (int i = 0; i < size; i++) {
    cout<<"----------------------------------------"<<endl;
    cout<<"Employee Id :"<<_emp[i]->getEmployeeId()<<endl;
    cout<<"Employee Name :"<<_emp[i]->getEmployeeName()<<endl;
    float salary = _emp[i]->calculateSalary();
    cout<<"Employee Salary :"<<salary<<endl;
    cout<<"----------------------------------------"<<endl;
  }
}
```

Output



Static vs Dynamic Binding

Static binding means that target function for a call is selected at compile time

Dynamic binding means that target function for a call is selected at run time

Static vs Dynamic Binding

```
HourlyEmployee he;
// Always HourlyEmployee::calcualteSalary called
he.calculateSalary();

Employee* _emp = new HourlyEmployee();
// Employee::calculateSalary called if not virtual
_emp->calculateSalary();
Employee * _emp = new HourlyEmployee();
// HourlyEmployee::calculateSalary() called if virtual
_emp->calculateSalary()
```

Then What is Term “Function Overriding” ?

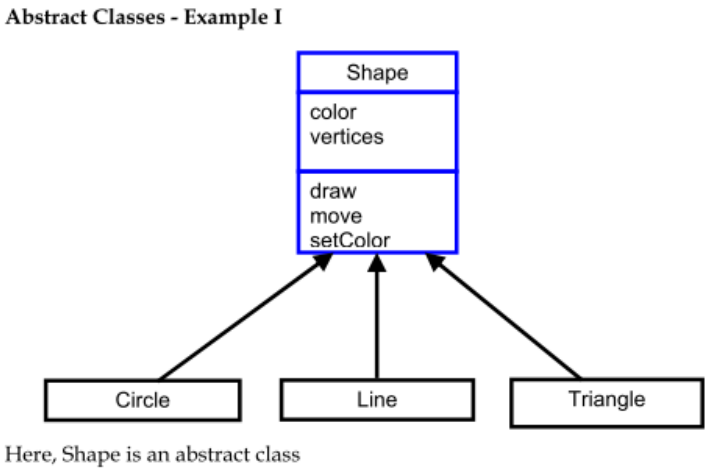
- If base class and derived class have member functions with same name and arguments.
- If you create an object of derived class and write code to access that member function then,
- the member function in derived class is only invoked, i.e.,
- the member function of derived class overrides the member function of base class.
- This feature in C++ programming is known as function overriding.
- Therefore calcualteSalary() is in child class as an overridden function

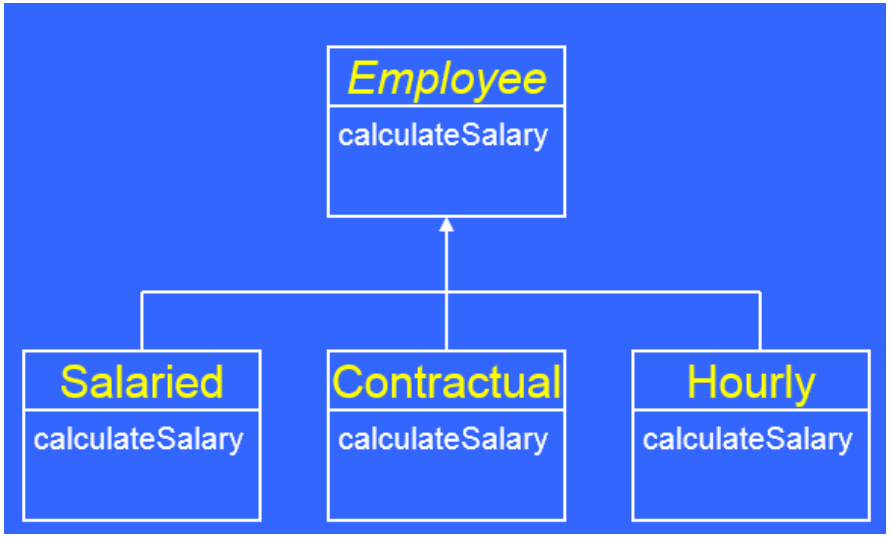
Function Overriding

- Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Abstract Class

- An abstract class implements an abstract concept
- Main purpose is to be inherited by other classes
- Can't be instantiated
- Promotes reuse





Concrete Class

- Implements a concrete concept
- Can be instantiated
- May inherit from an abstract class or another concrete class
- In C++, we can make a class abstract by making its function(s) pure virtual
- Conversely, a class with no pure virtual function is a concrete class

Pure Virtual Function

- A pure virtual represents an abstract behavior and therefore may not have its implementation (body)
- A function is declared pure virtual by following its header with “= 0”
- virtual float calculateSalary() = 0;

Pure Virtual Function Cont…

```
class Employee{
    protected:
       int employeeId;
       string employeeName;
       float salary;
   public :
    .
    .
  virtual void calculateSalary()=0;
};
Employee emp // ERROR
```

Pure Virtual Function Cont…

- A derived class of an abstract class remains abstract until it provides implementation for all pure virtual functions

Pure Virtual Function Cont…

```
class SalariedEmployee : public Employee{
 private:
   float houseRent;
    public:
 // No overridden calculateSalary()
};
SalariedEmployee em; // ERROR
```

Virtual Destructor

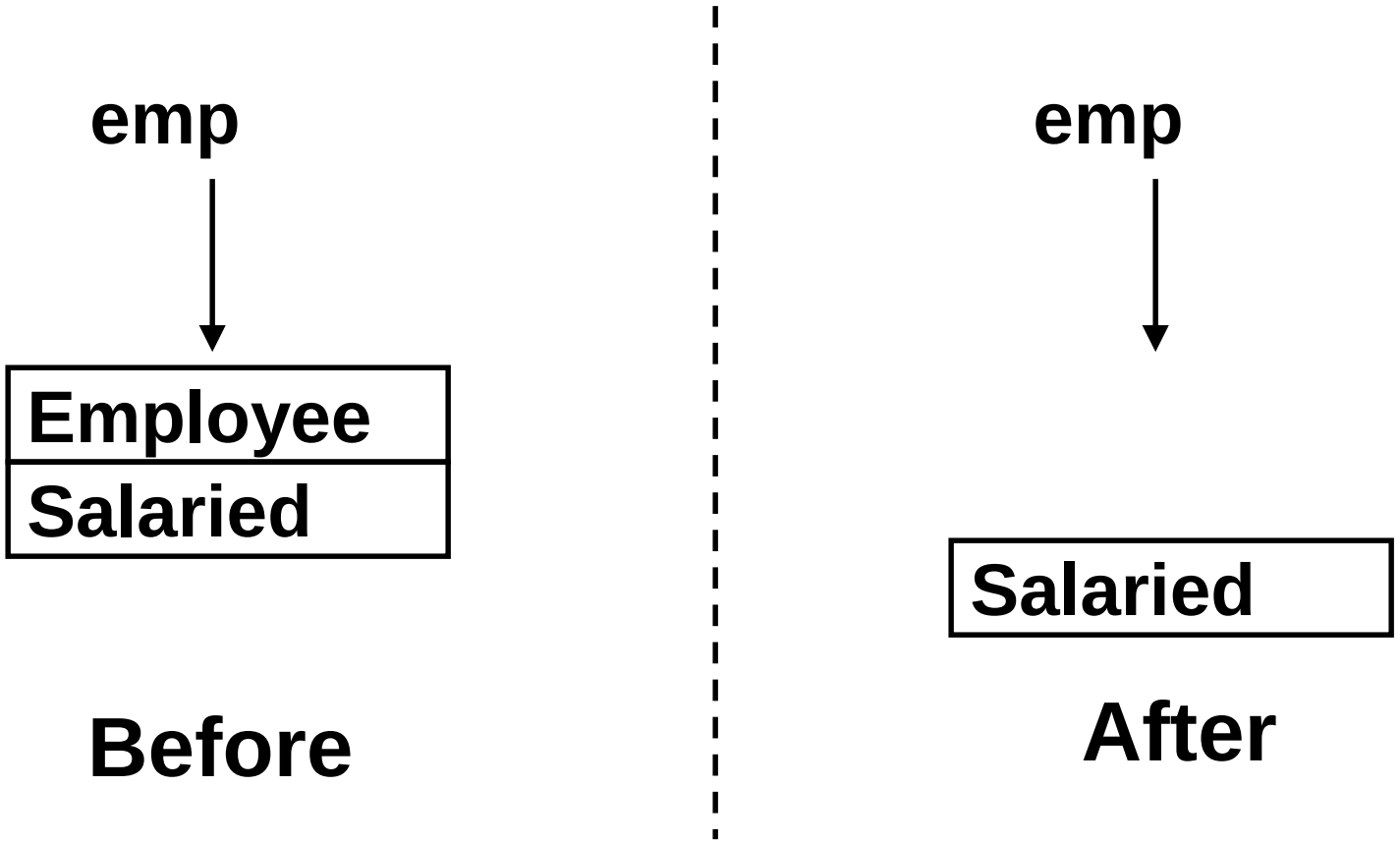
- When delete operator is applied to a **base** class pointer, **base** class destructor is called regardless of the object type

```
int main() {
        Employee* emp = new Salaried;
        delete emp;
        return 0;
}
```

Output

```
Employee destructor called
```

Virtual Destructor



Virtual Destructors

Make the base class destructor virtual

```
class Employee{
  …
  public:
  virtual ~Employee() {
    cout << "Employeedestructor
  called\n"; }
}
```

Now base class destructor will run after the derived class destructor

Virtual Destructor

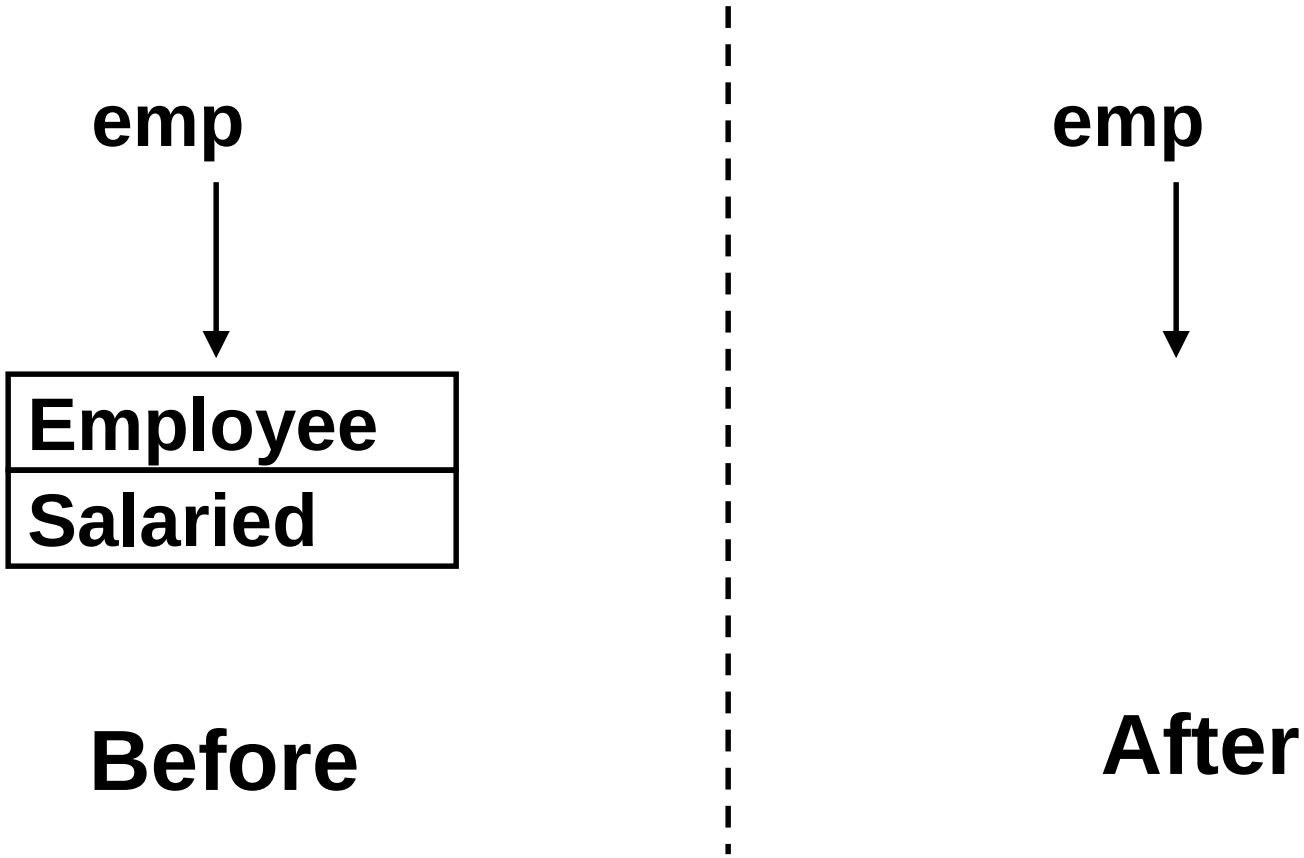
- When delete operator is applied to a **base** class pointer, **base** class destructor is called regardless of the object type

```
int main() {
        Employee* emp = new Salaried;
        delete emp;
        return 0;
}
```

Output

```
Salaried Employee destructor called
Employee destructor called
```

Virtual Destructor



Virtual Functions – Usage

Inherit interface and implementation

Just inherit interface (Pure Virtual)

V Table

Compiler builds a virtual function table (vTable) for each class having virtual functions

A vTable contains a pointer for each virtual function

Dynamic Dispatch

For non-virtual functions, compiler just generates code to call the function

In case of virtual functions, compiler generates code to

- access the object
- access the associated vTable
- call the appropriate function

Conclusion

Polymorphism adds

- Memory overhead due to vTables
- Processing overhead due to extra pointer manipulation

However, this overhead is acceptable for many of the applications

Moral: “Think about performance requirements before making a function virtual”