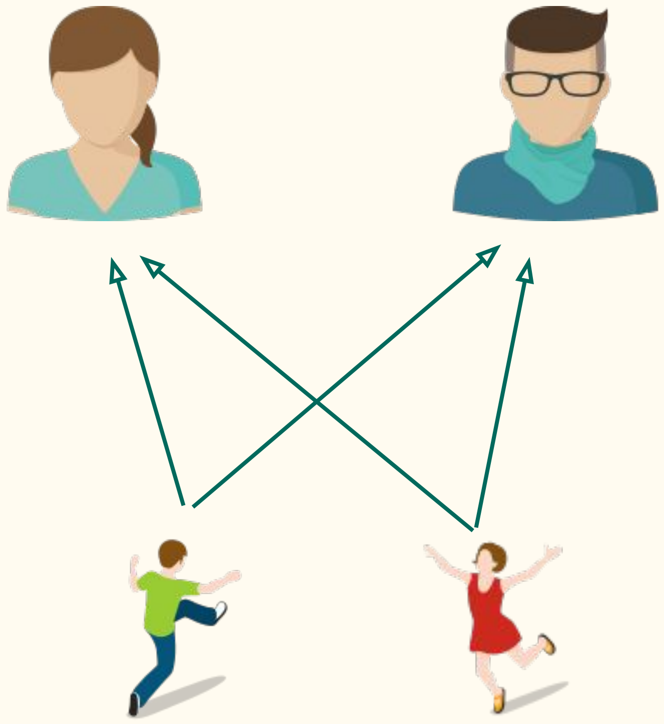
Chapter 6: Inheritance

Department of Computer Science and Engineering
Kathmandu University

Instructor: Rajani Chulyadyo, PhD

Contents

Inheritance



In biology, inheritance is the passing on of traits from parents to their offspring.

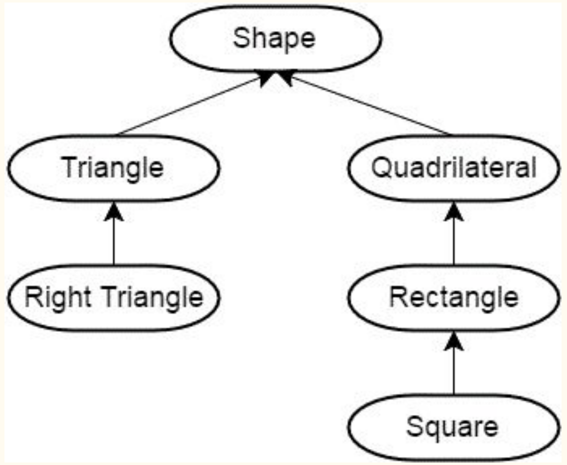
Similar notion in programming.

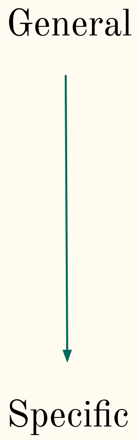
In programming, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.

Icons made by https://freeicons.io/profile/823 and https://freeicons.io/profile/722 from www.freeicons.io

Inheritance

- Models an “is-a” relationship between objects.
- Involves creating new objects by directly acquiring the attributes and behaviors of other objects and then extending or specializing them.





This diagram goes from general (top) to specific (bottom), with each item in the hierarchy inheriting the properties and behaviors of the item above it.

Source: https://www.learncpp.com/cpp-tutorial/111-introduction-to-inheritance/

Inheritance

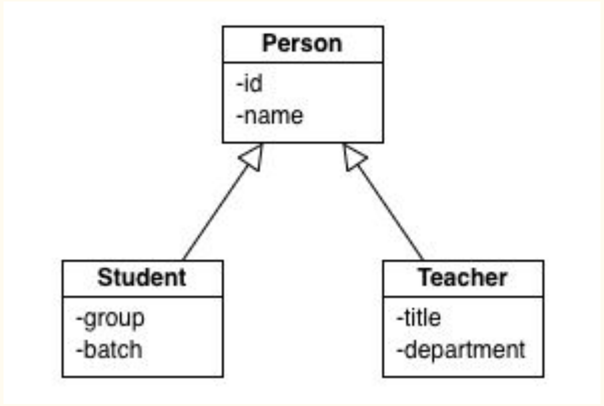
- Inheritance in C++ takes place between classes.
- In an inheritance (is-a) relationship, the class being inherited from is called the **parent class, base class,** or **superclass**, and the class doing the inheriting is called the **child class, derived class,** or **subclass.**
- A child class inherits both behaviors (member functions) and properties (member variables) from the parent (with different access permission).
- These variables and functions become members of the derived class.
- Child classes can have their own members that are specific to that class (specialization).

Inheritance

When you derive a class from another class, the new class gets all the functionality of the base class plus whatever new features you add.

You can add data members and functions to the new class but you cannot remove anything from what the base class offers.

Inheritance



`Person` is a base class.

`Person` is inherited (or extended) by child classes, `Student` and `Teacher`.

In UML class diagrams, inheritance (also called generalization) is indicated by a triangular arrowhead on the line connecting the parent and child classes.

Inheritance

Its big payoff is that it permits code reusability.

A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Example

Access specifier

Access level

```
// Base class
class Person {
   public:
      long id;
      std::string name;

      std::string getName() const
         { return name; }
};
```

```
// Derived class
class Student : public Person {
   public:
      std::string group;
      std::string batch;

      std::string getGroup()
const
         { return group; }
};
```

Inheritance in C++

Syntax

`class` *`DerivedClassName`* `:` <u>`access-level`</u> *`BaseClassName`*

`access-level` (aka visibility mode, derivation type) specifies the type of derivation/inheritance, and it can be

- private (by default)
- protected or
- public

Visibility mode: public

When a base class is publicly inherited by a derived class,

- Private members of the base class are not inherited and therefore will never become the members of its derived class
- **Public** members of the base class become **public** members of the derived class and therefore are accessible to the objects of the derived class.

Example

Private members will not be inherited

Public members of the base class will be inherited as public members of the derived class.

```
// Base class
class Person {
   private:
       long id;
   public:
       std::string name;
       std::string getName() const
           { return name; }
};
```

```
// Derived class
class Student : public Person {
   private:
       std::string group;
       std::string batch;
   public:
       void print() const
           {  std::cout << name // public
               << ": " << group << "\n"; }
};
int main() {
   Student s;
   s.name = "Name"; // name is public
   s.print();
   s.getName(); // getName() is public
}
```

Visibility mode: private

When a base class is privately inherited by a derived class,

- Private members of the base class are not inherited and therefore will never become the members of its derived class
- **Public** members of the base class become **private** members of the derived class and therefore can only be accessed by the member functions of the derived class.

Example

Private members will not be inherited

Public members of the base class will be inherited as private members of the derived class.

```cpp
// Base class
class Person {
  private:
    long id;
  public:
    std::string name;
    std::string getName() const
      { return name; }
};
```

```cpp
// Derived class
class Student : private Person {
  private:
    std::string group;
    std::string batch;
  public:
    void print() const
      {  std::cout << name // private
         << ": " << group << "\n"; }
};
int main() {
  Student s;
  s.name = "Name"; // Error! name is private
  s.print();
  s.getName(); // Error! getName() is private
}
```

The `protected` access-specifier

- Protected members are like private members.
  - They are accessible by the member functions within the class but
  - are not accessible from outside the class
- The difference between private members and protected members is that unlike private members, protected members are also accessible from any class immediately derived from it.
- The protected access-specifier allows the derived classes to access the member.

Visibility mode: protected

- When a class is inherited in public mode, protected members of the base class become protected in the derived class.
- When a class is inherited in private mode, protected members of the base class become private in the derived class.
- When a class is inherited in protected mode, both public and protected members of the base class become protected members of the derived class.

Example

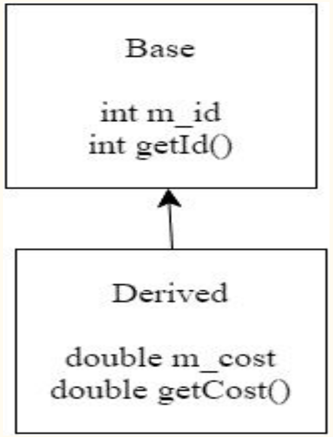
Access specifier

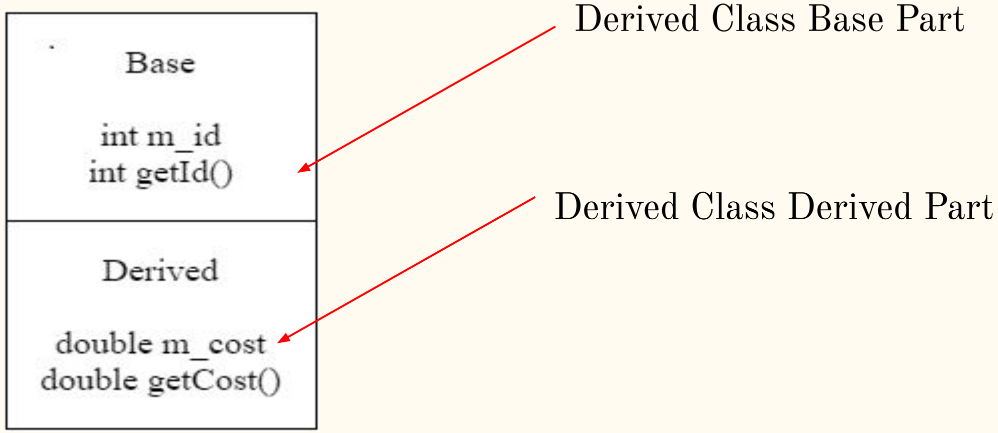
```cpp
// Base class
class Person {
  protected:
      long id;
      std::string name;
  public:
      std::string getName() const
         { return name; }
};
```

```cpp
// Derived class
class Student : public Person {
   private:
        std::string group;
        std::string batch;
   public:
        void print() const
            {  std::cout << name
               << ": " << group << "\n"; }
};
int main() {
   Student s;
   s.print();
   s.getName();
}
```

Order of Construction of derived class

- When we instantiate an instance of Derived, first Base portion of Derived is constructed using the Base Default Constructor.
- Once Base portion is finished, the Derived portion is constructed using the Derived Default Constructor.





Example

```
class Base
{
public:
    int m_id;
    Base(int id=0)
        : m_id(id)
    {
        std::cout << "Base\n";
    }
    int getId() const { return m_id; }
};
class Derived: public Base
{
public:
    double m_cost;
    Derived(double cost=0.0)
        : m_cost(cost)
    {
        std::cout << "Derived\n";
    }
   double getCost() const { return m_cost; }
};
```

```
int main()
{
    std::cout << "Instantiating Base\n";
    Base cBase;
    std::cout << "Instantiating Derived\n";
    Derived cDerived;
    return 0;
}
```

Output of the program will be like this

```
Instantiating Base
Base
Instantiating Derived
Base
Derived
```

Order of Construction of Derived Class

Order of Construction for Inheritance Chain

- It is sometimes the case that classes are derived from other classes, which are themselves derived from other classes.
- C++ always constructs the “first” or “most base ” class first. It then walks through the inheritance tree inorder and constructs each successive derived class.

Example

```
class A
{
public:
    A(){ std::cout << "A\n";}
};

class B: public A
{
public:
    B() { std::cout << "B\n";}
};

class C: public B
{
public:
    C() { std::cout << "C\n";}
};
```

```
int main()
{
    std::cout << "Constructing A: \n";
    A cA;

    std::cout << "Constructing B: \n";
    B cB;

    std::cout << "Constructing C: \n";
    C cC;
}
```

Order of Construction of Derived Class

- In the case of the default constructor, it is implicitly available from Parent to Child Class.
- Parameterized constructor are not accessible to the derived class automatically.
- Explicit call has to be made in the child class constructor for accessing the parameterized constructor of the Parent Class to the Child Class.

Function overriding

Redefinition of base class function in its derived class with same signature, i.e return type and parameters.

Overriding is needed when derived class function has to do some added or different job than the base class function.

Function overriding

Example:

```cpp
class Point
{
protected:
   double x, y;
public:
   Point(double x = 0, double y = 0)
       : x(x), y(y) {}
   void print() {
       std::cout << "(" << x << ", " << y
<< ")";
   }
};
```

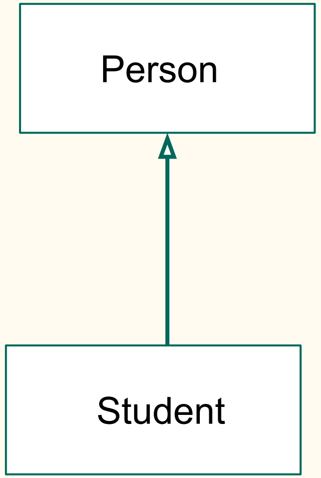
```cpp
class Point3D : public Point
{
protected:
   double z;
public:
   Point3D(double x = 0, double y = 0,
double z = 0)
       : Point(x, y), z(z) {}

   // Overriding print() function
   void print() {
       std::cout << "(" << x << ", " << y <<
", " << z << ")";
   }
};
```

Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

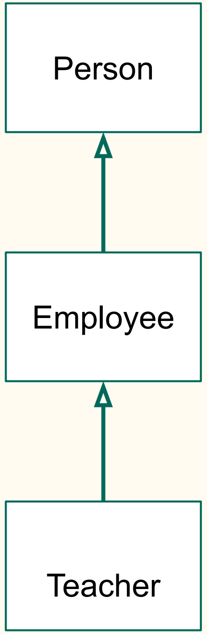
Types of Inheritance: Single Inheritance

- A class derived from only one class.
- Student is a Person, so class Student is derived from class Person.



Types of Inheritance: Multilevel Inheritance

- A class derived from another derived class
- Class `Teacher` is derived from class `Employee` and class `Employee` is derived from another class `Person`.



Example

```
class Person {
protected:
    std::string name;
    int age;
public:
  Person(){}
  Person(std::string name,int age)
    :name(name),age(age){}

  void print() const{
    std::cout << "Name: " << name
      << " Age: " << age << " ";
  }
};
```

```
class Employee : public Person {
protected:
    std::string employer;
    double wage;
public:
    Employee(){}
    Employee(std::string name, int age,
        std::string employer, double wage)
       :Person(name,age), employer(employer),
        wage(wage){}

    void print() const {
        Person::print();
        std::cout << "Employer:" << employer
            << " Wage:" << wage << " ";
    }
};
```

Example

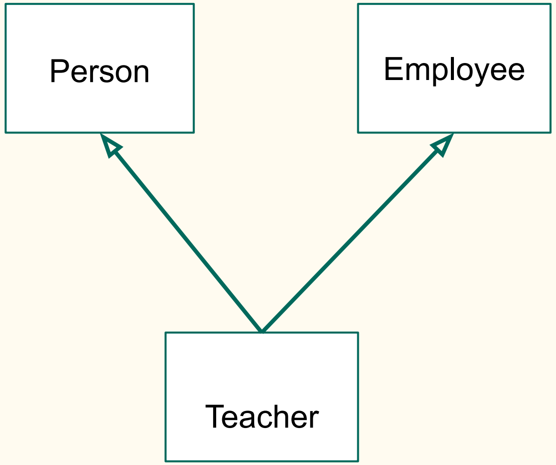
Employee extends Person

```
class Teacher : public Employee {
protected:
    int teachergrade;
public:
    Teacher(){}
    Teacher(std::string name, int age,
            std::string employer,
            double wage, int teachergrade)
      :Employee(name, age, employer, wage),
       teachergrade(teachergrade){}

 void print() const{
   Employee::print();
   std::cout << "TeachersGrade: "
       << teachergrade;
  }
};
```

```
int main(){
    Teacher t("ram", 30, "test",
               12345, 4);
    t.print();
}
```

Types of Inheritance: Multiple Inheritance

- A class derived from more than one base class
- A teacher is a Person and Employee as well so class `Teacher` is derived from two classes `Person` and `Employee`.



Example

```
class Person{
protected:
    std::string name;
    int age;
public:
    Person(){std::cout<<"default
Person"<<std::endl;}
    Person(std::string name,int age)

:name(name),age(age){std::cout<<"param
person"<<std::endl;}
    void print_person()const{
    std::cout<<"Name:"<<name<<"
"<<"Age:"<<age<<" ";}

};
```

```
class Employee{
protected:
    std::string employer;
    double wage;
public:
    Employee(){std::cout<<"default
Employee"<<std::endl;}
    Employee(std::string employer,double wage)

:employer(employer),wage(wage){std::cout<<"param
employee"<<std::endl;}
    void print_employee()const{
     std::cout<<"Employer:"<<employer<<"
"<<"Wage:"<<wage<<" ";
    }

};
```

Example

```
class Teacher:public Employee,protected Person{
protected:
    int teachergrade;
public:
    Teacher(){std::cout<<"default Teacher"<<std::endl;}
    Teacher(std::string name,int age,std::string
employer,double wage,int teachergrade)
:Person(name,age),Employee(employer,wage),teachergrade
(teachergrade){std::cout<<"param teacher"<<std::endl;}

 void print_teacher() const{
    print_person();
    print_employee();
    std::cout<<"TeachersGrade:"<<teachergrade;
  }
};
```

```
int main(){
    Teacher t("ram",30,"test",12324,4);
    t.print_teacher();
    return 0;
}
```

Types of Inheritance: Multiple Inheritance

Problems with multiple Inheritance

- Ambiguity can result when multiple base classes contain a function with a same name.

Example

```cpp
class Person{
protected:
    std::string name;
    int age;
public:
    Person()
    { std::cout << "default Person\n"; }
    Person(std::string name,int age)
    :name(name),age(age)
    {  std::cout << "param person\n";   }

    void print()const
    {  std::cout << "Name:  " << name
          << " Age:" << age << " ";
    }
};
```

```cpp
class Employee{
protected:
    std::string employer;
    double wage;
public:
    Employee()
    {  std::cout<<"default Employee\n";  }
    Employee(std::string employer,double wage)
      : employer(employer), wage(wage)
      {  std::cout  <<  "param employee\n";  }

    void print()const
    {  std::cout << "Employer:" << employer
          << " Wage:" << wage << " ";
    }
};
```

Example

```cpp
class Teacher : public Employee, public Person {
protected:
    int teachergrade;
public:
    Teacher()
    {   std::cout << "default Teacher\n";   }
    Teacher(std::string name, int age,
            std::string employer, double wage,
            int teachergrade)
   : Person(name,age), Employee(employer,wage),
     teachergrade(teachergrade)
     {   std::cout<<"param teacher\n";   }
};
```

```cpp
int main(){
    Teacher t("ram",30,"test",12324,4);
    t.print(); //error Teacher::print()
               // is ambiguous

    //call employee version of print()
    t.Employee::print();

    //call Person version of print()
    t.Person::print();

    return 0;
}
```

Types of Inheritance: Multiple Inheritance

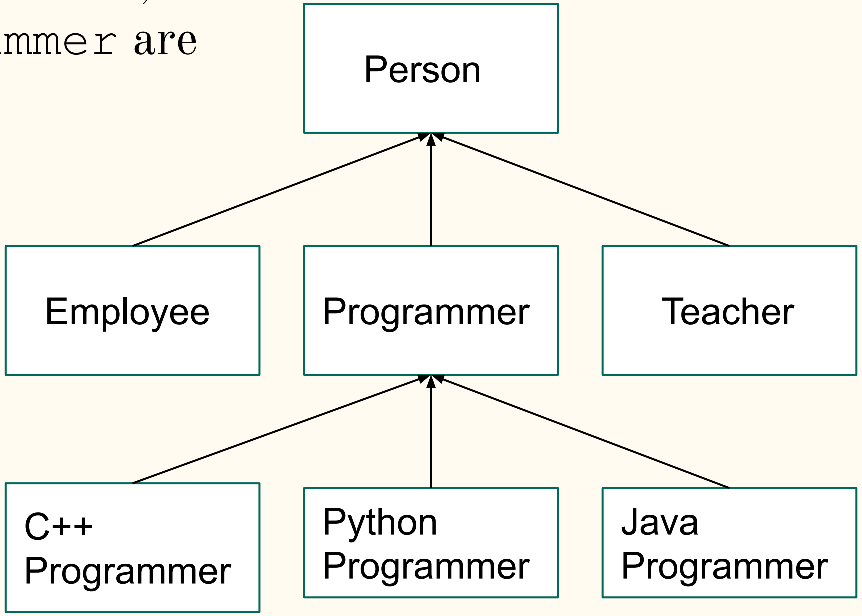
- When `t.print()` is compiled, the compiler looks to see if `Teacher` contains a function named `print()`. It doesn't.
- The compiler then looks to see if any of the parent classes have a function named `print()`
- The problem is that object t actually contains Two `print()` functions:
  - One inherited from `Person`
  - One inherited from `Employee`
- However, there is a way to work around this problem. We can explicitly specify which version we meant to call.

E.g `t.Person::print() //call Person version of getInfo()`

Source: https://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/

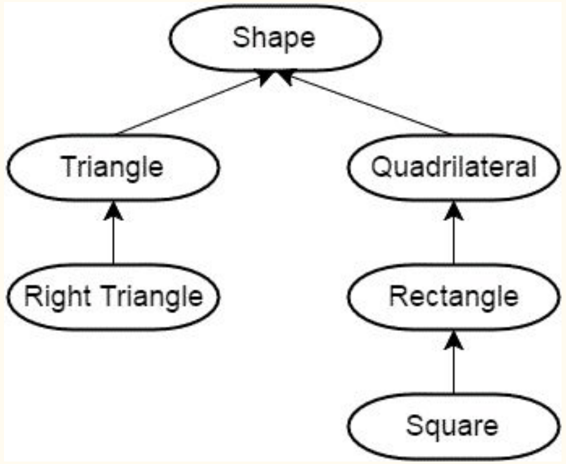
Types of Inheritance: Hierarchical Inheritance

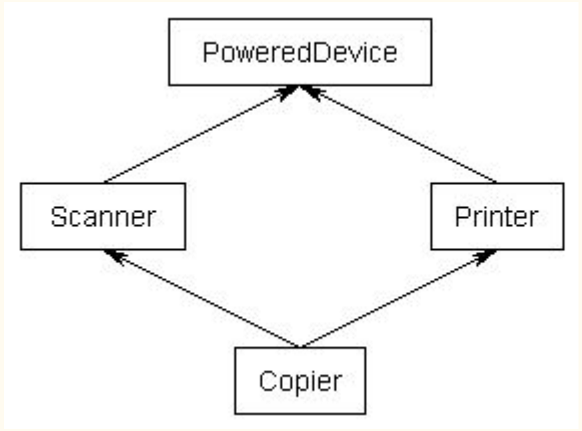
- More than one class are derived from Single Base class.
- Employee, Teacher, Programmer all are Person, so class `Employee`, `Teacher` and `Programmer` are derived from class `Person`.



Types of Inheritance: Hybrid Inheritance

- Combination of two or more Inheritance





Lab 5

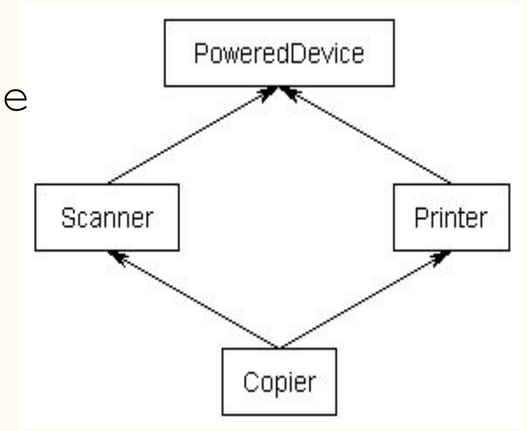
Question 1:

Create a class called Polygon with two data members: numberOfSides, and centroid (a Point object, you may use the Point class from previous lectures) and two member functions: display() that displays the values of member variables, and move() that translates the Polygon object to a new location.

Create two other classes Triangle and Rectangle inheriting from Polygon class. Add relevant data members and member functions in these classes.

The Diamond Problem

- **Diamond Problem** occurs when two superclasses of a class have a common base class.
- Here, `Copier` class would get the two copies of `PoweredDevice` class. One from `Printer` and one from `Scanner`.
- `PoweredDevice` gets constructed twice.
- When any data/member function of class `PoweredDevice` is accessed by an object of class `Copier`, ambiguity arises as which data/member function would be called.



Example

```
#include <iostream>
class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
     std::cout << "PoweredDevice: "<< power << '\n';
    }
};
```

Example

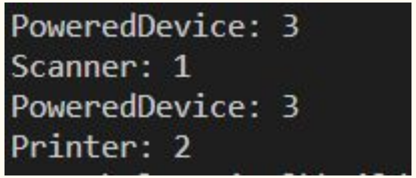
```
class Scanner : public PoweredDevice
{
public:
    Scanner(int scanner, int power)
        : PoweredDevice(power)
    {
     std::cout << "Scanner: " << scanner
         << '\n';
    }
};
```

```
class Printer : public PoweredDevice
{
public:
    Printer(int printer, int power)
        : PoweredDevice(power)
    {
     std::cout << "Printer: " << printer
         << '\n';
    }
};
```

Example

```
class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power)
        : Scanner(scanner, power),
          Printer(printer, power)
    {   }
};

int main()
{
    Copier copier(1, 2, 3);
}
```

- In this example, `PoweredDevice` would be constructed twice when we create `Copier` class object.
- So the output of the program will be like this.



```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

Virtual Base Classes

- To resolve this ambiguity (diamond problem), common base class has to be declared as virtual base class by inserting the "`virtual`" keyword in the inheritance list of the derived class.
- Virtual Base Class means there is only one base object.
- The base object is shared between all objects in the inheritance tree and is only constructed once.

```
class PoweredDevice
{  };

class Scanner: virtual public PoweredDevice
{  };
```

```
class Printer: virtual public PoweredDevice
{  };

class Copier: public Scanner, public
Printer
{  };
```

Virtual Base Classes

- Virtual base classes are always created before non-virtual base classes.
- If a class inherits one or more classes that have virtual parents, the most derived class is responsible for constructing the virtual base class

Abstract class

- An abstract class is one that is not used to create objects.
- It is designed only to act as a base class.
- Abstract class cannot be instantiated but pointers and reference of abstract class type can be created.
- A class is Abstract class when it has at least one pure virtual function.
- Pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.
- To create a pure virtual function rather than define a body for the function, we simply assign the function the value 0.

```
virtual int getValue() const = 0;
```

Assignment

Assignment # 3

1. What is Operator Overloading? Why it is important?
2. What are the different ways through which operator can be overloaded?
3. Is it possible to overload the I/O operator using member function? Explain.
4. What is the difference between copy constructor and copy assignment operator?
5. What is inheritance? What are the advantages of inheritance?
6. Why is the `protected` access specifier needed?
7. In what order are the class destructors called when a derived class object is destroyed?
8. What are the differences between function overloading and function overriding?