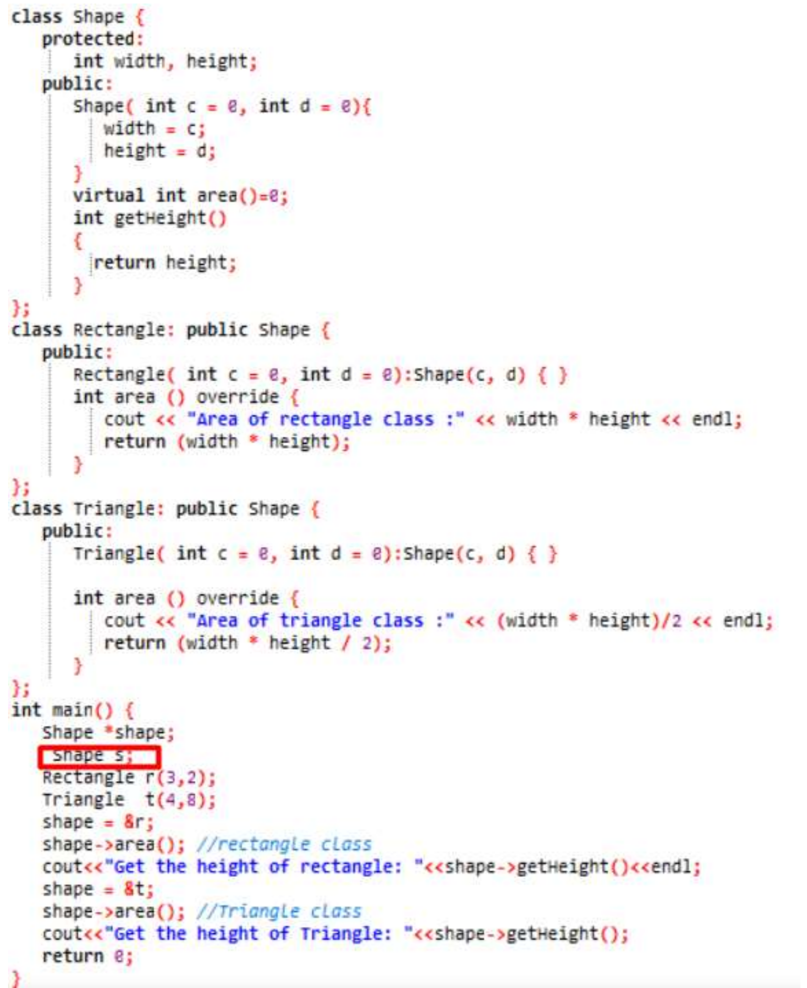
Lecture 15
OOP

Khola Naseem
khola.naseem@uet.edu.pk

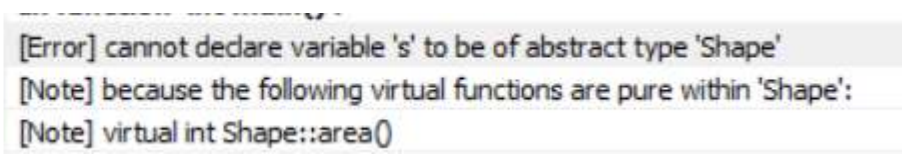
Abstract Class:

- A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.
- We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions).
-



```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area()=0;
      int getHeight()
      {
         return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Shape s;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get the height of Triangle: "<<shape->getHeight();
   return 0;
}
```

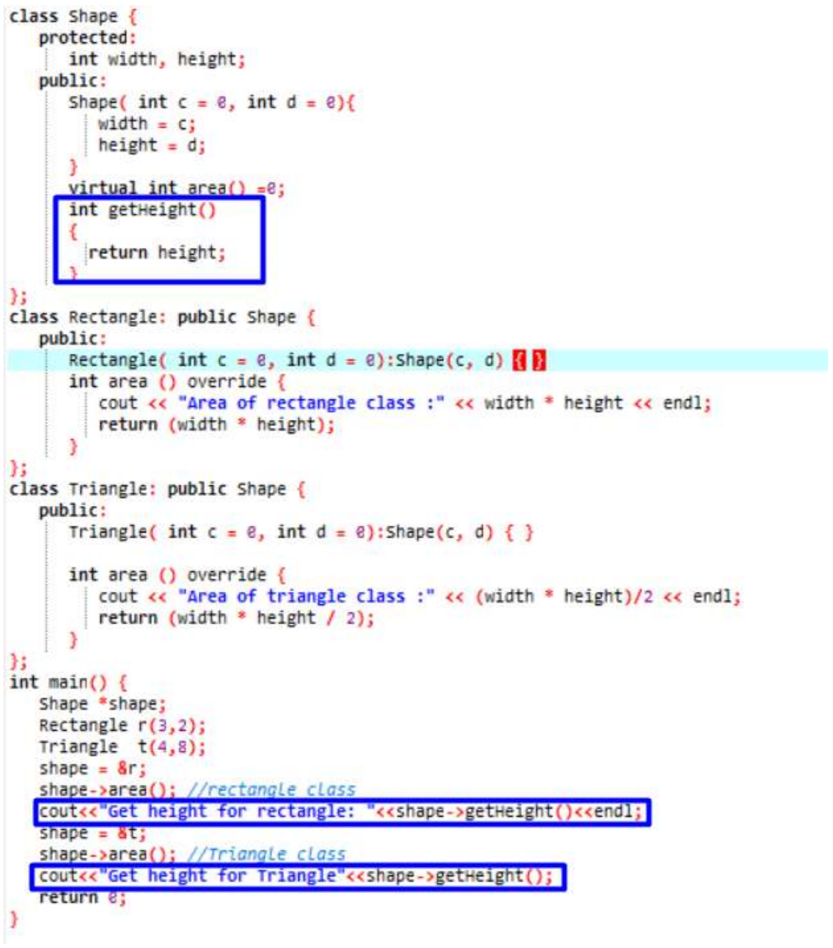
Error



```
[Error] cannot declare variable 's' to be of abstract type 'Shape'
[Note] because the following virtual functions are pure within 'Shape':
[Note] virtual int Shape::area()
```

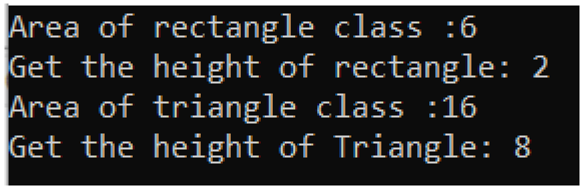
Credit: Khola Naseem

Abstract Class:

- A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.
- We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions) .
- 

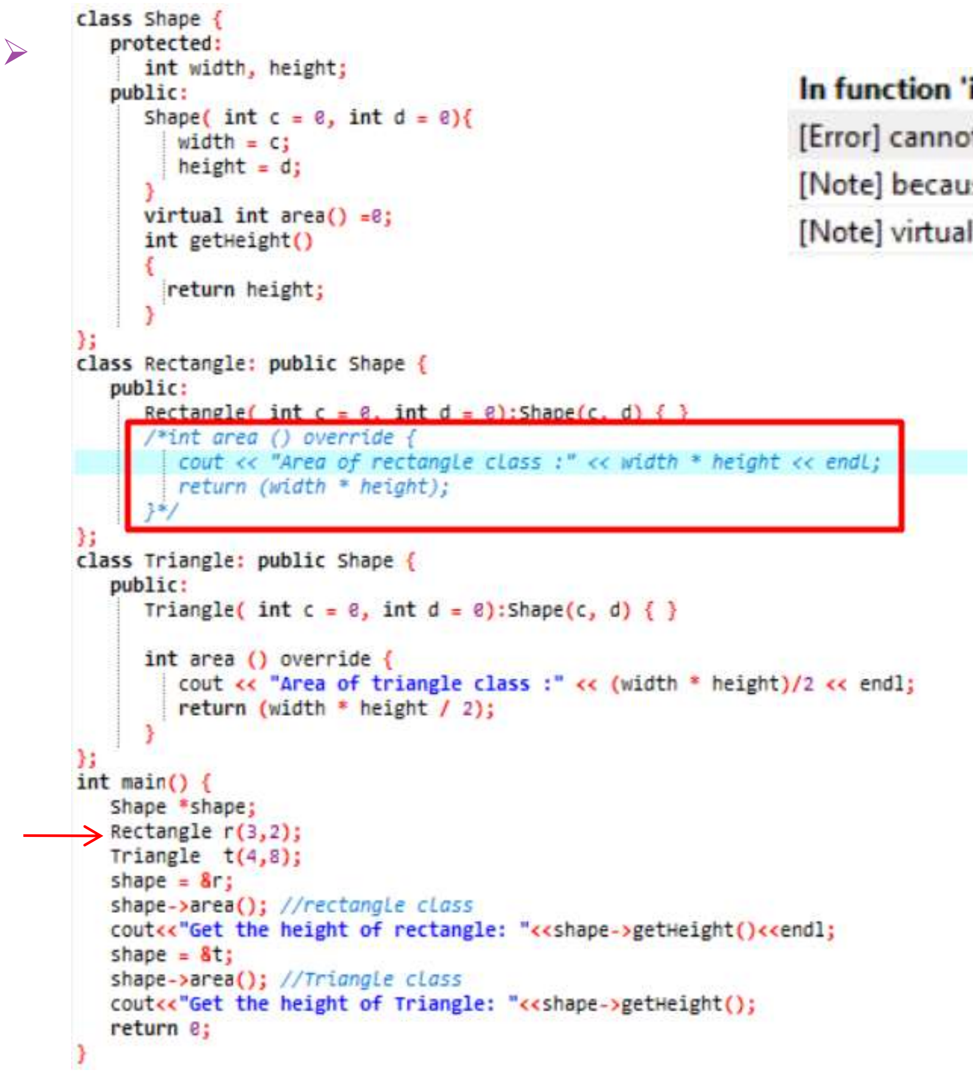
```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int c = 0, int d = 0){
      width = c;
      height = d;
    }
    virtual int area() =0;
    int getHeight()
    {
      return height;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int c = 0, int d = 0):Shape(c, d) { }
    int area () override {
      cout << "Area of rectangle class :" << width * height << endl;
      return (width * height);
    }
};
class Triangle: public Shape {
  public:
    Triangle( int c = 0, int d = 0):Shape(c, d) { }

    int area () override {
      cout << "Area of triangle class :" << (width * height)/2 << endl;
      return (width * height / 2);
    }
};
int main() {
  Shape *shape;
  Rectangle r(3,2);
  Triangle  t(4,8);
  shape = &r;
  shape->area(); //rectangle class
  cout<<"Get height for rectangle: "<<shape->getHeight()<<endl;
  shape = &t;
  shape->area(); //Triangle class
  cout<<"Get height for Triangle"<<shape->getHeight();
  return 0;
}
```

Output:

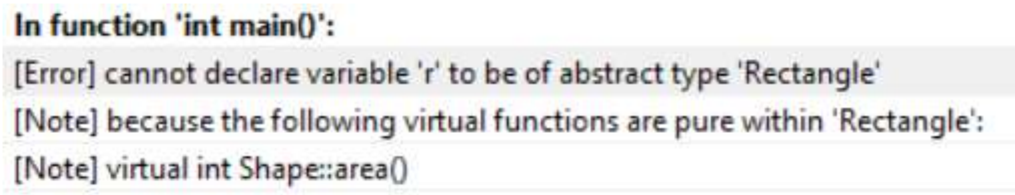


```
Area of rectangle class :6
Get the height of rectangle: 2
Area of triangle class :16
Get the height of Triangle: 8
```

Abstract Class:

- In this case, we can create a pure virtual function named area() in the Shape. Since it's a pure virtual function, all derived classes Triangle and Rectangle must include the area() function with implementation. otherwise, the derived class is also abstract.
- 

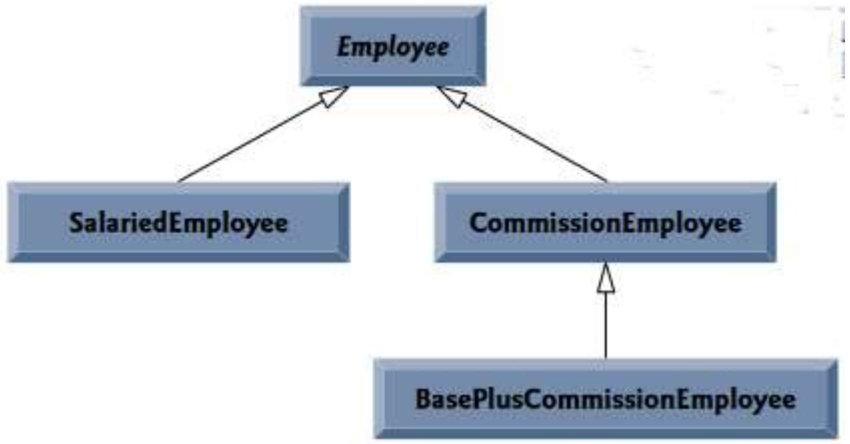
```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area() =0;
      int getHeight()
      {
        return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      /*int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }*/
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get the height of Triangle: "<<shape->getHeight();
   return 0;
}
```



```
In function 'int main()':
[Error] cannot declare variable 'r' to be of abstract type 'Rectangle'
[Note] because the following virtual functions are pure within 'Rectangle':
[Note] virtual int Shape::area()
```

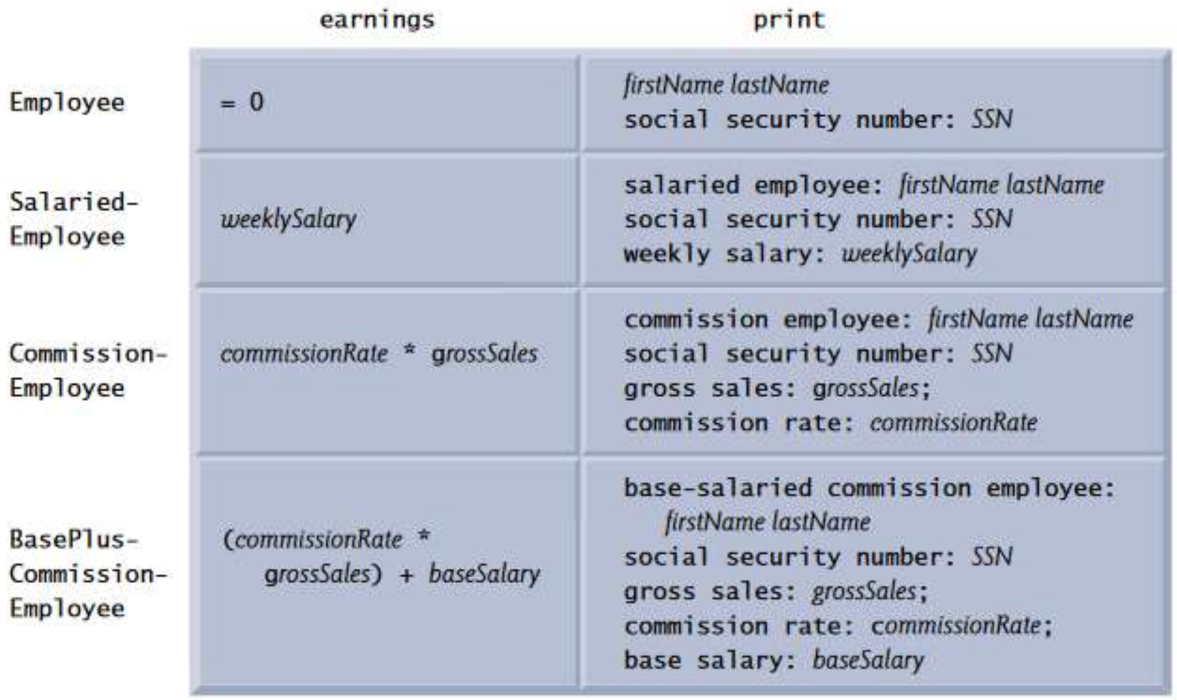
Abstract Class:

- **Example:**
- Creating Abstract Base Class Employee
- Class Employee provides functions earnings and print, in addition to various get and set functions that manipulate Employee's data members. An earnings function certainly applies generally to all employees, but each earnings calculation depends on the employee's class. So we declare earnings as pure virtual in base class Employee because a default implementation does not make sense for that function—there is not enough information to determine what amount earnings should return. Each derived class overrides earnings with an appropriate implementation. To calculate an employee's earnings.



Abstract Class:

- **Example:**
- Including earnings as a pure virtual function in Employee forces every direct derived class of Employee that wishes to be a **concrete class** to override earnings
- Function print in class Employee displays the first name, last name and social security number of the employee. each derived class of Employee overrides function print to output the employee’s type (e.g., "salaried employee:") followed by the rest of the employee’s information



	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	*weeklySalary*	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*
Commission-Employee	*commissionRate* * *grossSales*	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(*commissionRate* * *grossSales*) + *baseSalary*	base-salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Credit: Khola Naseem

Abstract Class:

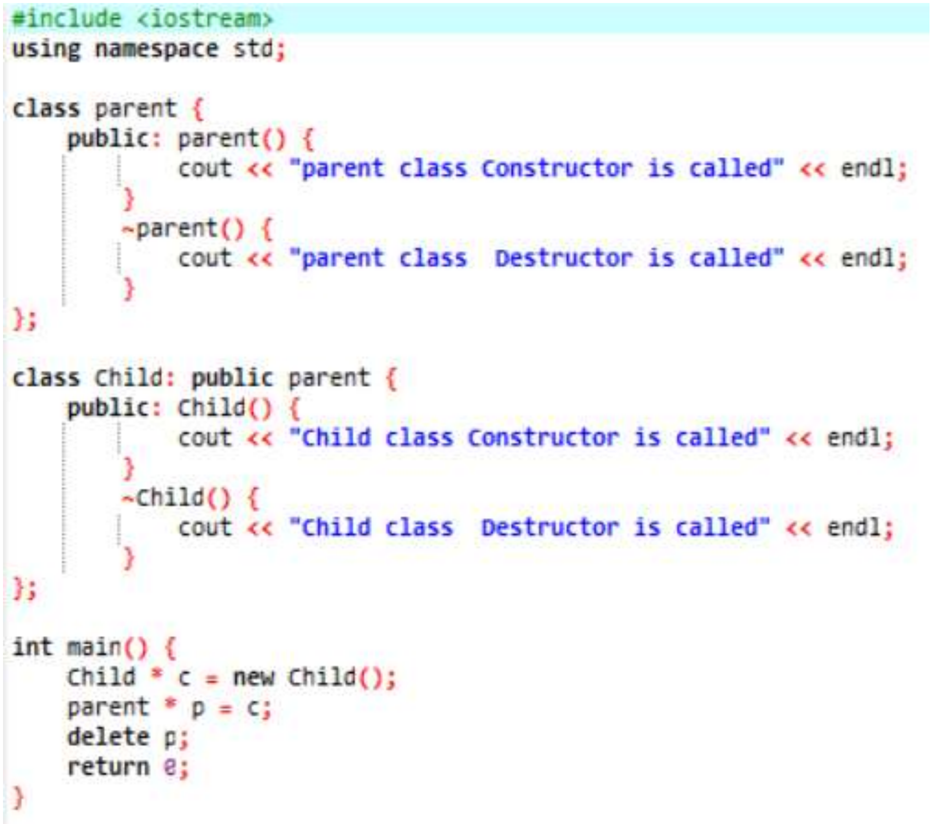
- An abstract class is a base class from which other classes can inherit.
- classes that can be used to instantiate objects are called concrete classes. Such classes define or inherit implementations for every member function they declare.
- We could have an **abstract base** class TwoDimensionalShape and derive such concrete classes as Square, Circle and Triangle. We could also have an abstract base class ThreeDimensionalShape and derive such **concrete classes** as Cube, Sphere and Cylinder.
- Abstract base classes are too generic to define real objects; we need to be more specific before we can think of instantiating objects. For example, if someone tells you to “draw the two-dimensional shape,” what shape would you draw?
- **Concrete classes** provide the specifics that make it possible to instantiate objects.

Virtual destructor:

- Virtual Destructor in C++ is used to release or free the memory used by the child class (derived class) object when the child class object is removed from the memory using the parent class's pointer object.
- **Virtual destructors** maintain the hierarchy of calling destructors from child class to parent class as the virtual keyword used in the destructor follows the concept of late binding or the **run-time binding.**
- The destructor of the parent class uses a virtual keyword before its name and makes itself a virtual destructor to ensure that the destructor of both the parent class and child class should be called at the run time.
- The derived class's destructor is called first, and then the parent class or base class releases the memory occupied by both destructors.

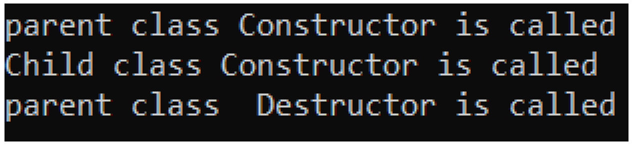
Virtual destructor:

- A destructor is implicitly invoked when an object of a class goes out of scope or the object's scope ends to free up the memory occupied by that object.
- Due to early binding, when the object pointer of the parent class is deleted, which was pointing to the object of the derived class then, only the destructor of the parent class is invoked; it does not invoke the destructor of the child class, which leads to the problem of memory leak in our program.



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

class parent {
    public: parent() {
            cout << "parent class Constructor is called" << endl;
        }
        ~parent() {
            cout << "parent class  Destructor is called" << endl;
        }
};

class Child: public parent {
    public: Child() {
            cout << "Child class Constructor is called" << endl;
        }
        ~Child() {
            cout << "Child class  Destructor is called" << endl;
        }
};

int main() {
    Child * c = new Child();
    parent * p = c;
    delete p;
    return 0;
}
```

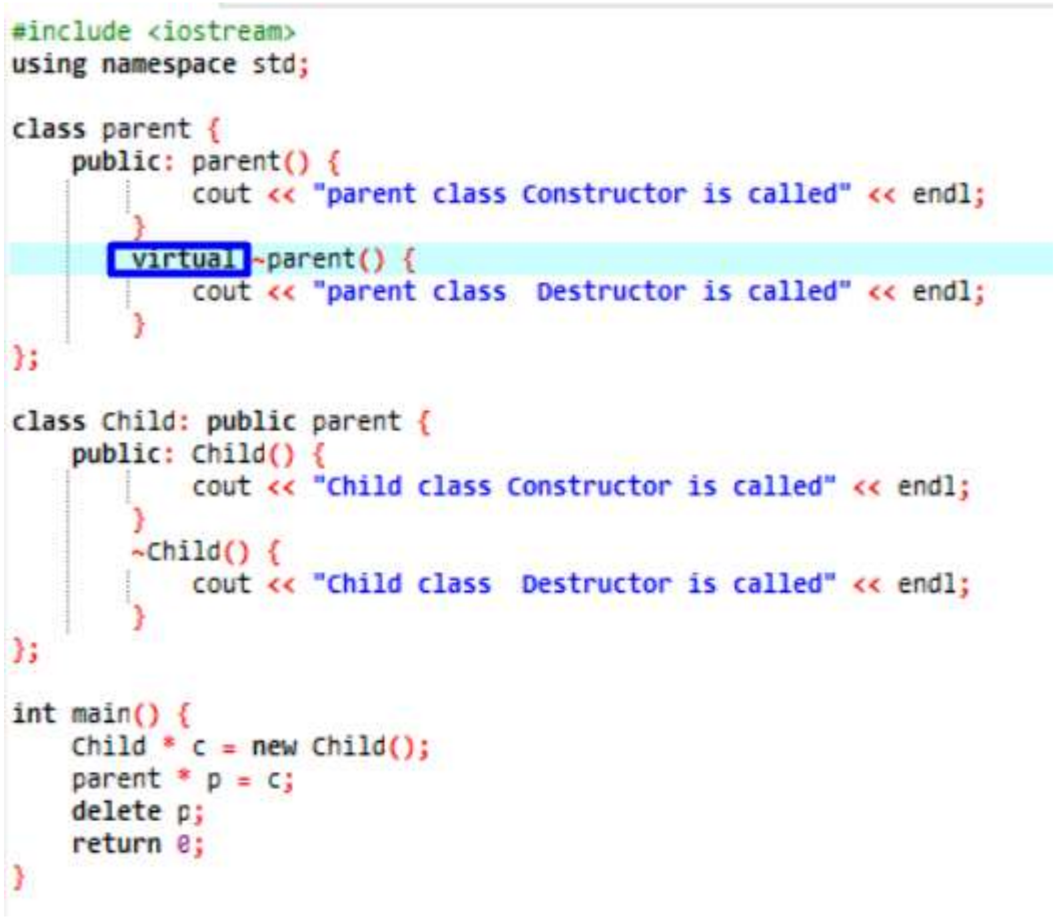
Output:



```
parent class Constructor is called
Child class Constructor is called
parent class  Destructor is called
```

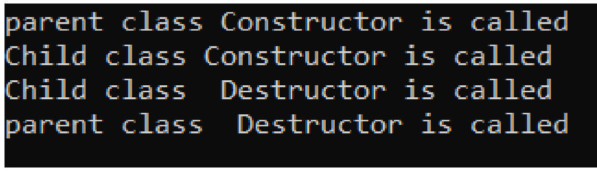
Credit: Khola Naseem

Virtual destructor:

- Virtual destructor, inside the parent class ensures that first the child class's destructor should be invoked. And then, the destructor of the parent class is called so that it releases the memory occupied by both destructors.
- 

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

class parent {
    public: parent() {
            cout << "parent class Constructor is called" << endl;
        }
        virtual ~parent() {
            cout << "parent class  Destructor is called" << endl;
        }
};

class Child: public parent {
    public: Child() {
            cout << "Child class Constructor is called" << endl;
        }
        ~Child() {
            cout << "Child class  Destructor is called" << endl;
        }
};

int main() {
    Child * c = new Child();
    parent * p = c;
    delete p;
    return 0;
}
```

Output:

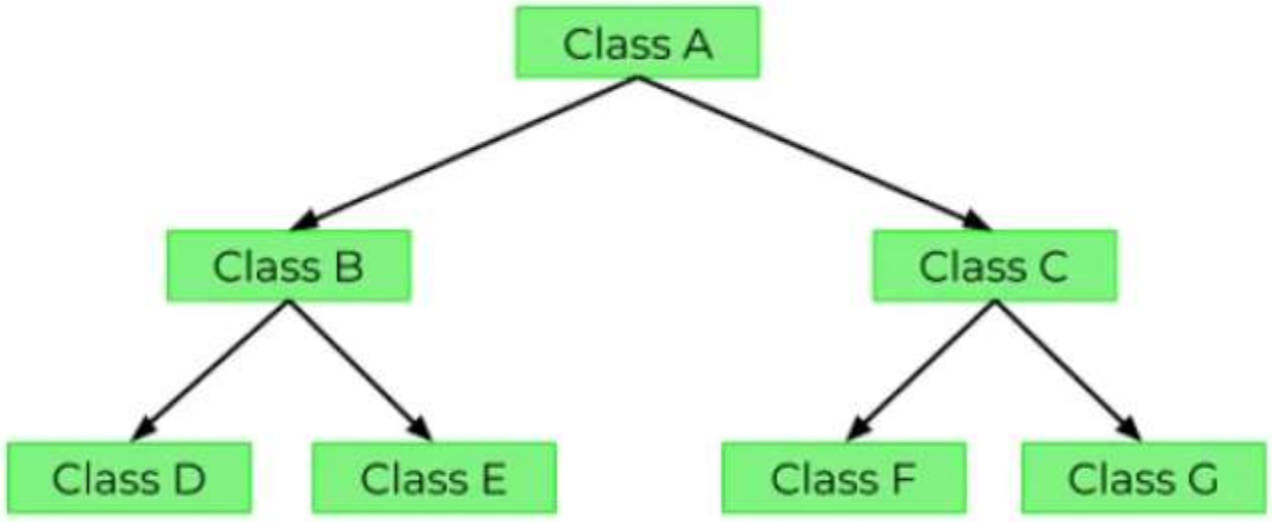


```
parent class Constructor is called
Child class Constructor is called
Child class  Destructor is called
parent class  Destructor is called
```

Credit: Khola Naseem

Hierarchical Inheritance in C++ :

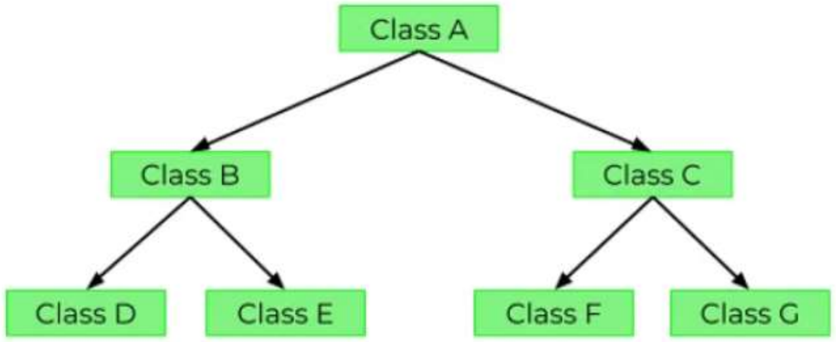
- If more than one class is inherited from the base class, it's known as hierarchical inheritance.
- **C++ Multilevel Inheritance:**
  - A single base class can have multiple derived classes, and other subclasses can further inherit these derived classes, forming a hierarchy of classes.



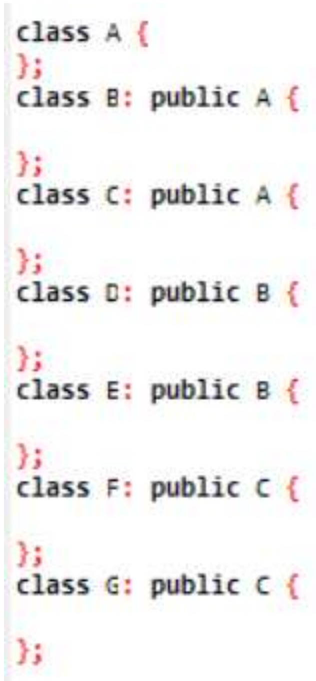
Credit: Khola Naseem

Hierarchical Inheritance in C++ :

- **C++ Multilevel Inheritance:**



- Implementation:

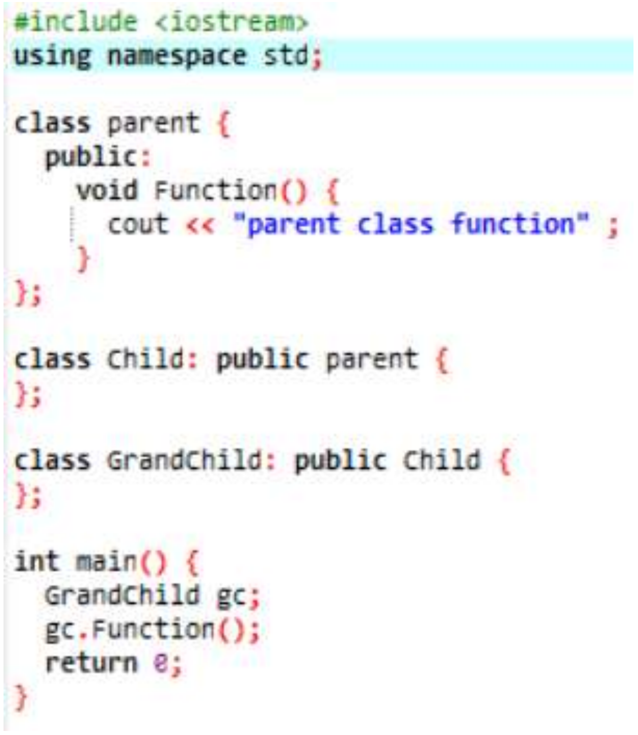


```
class A {
};
class B: public A {

};
class C: public A {

};
class D: public B {

};
class E: public B {

};
class F: public C {

};
class G: public C {

};
```

Credit: Khola Naseem

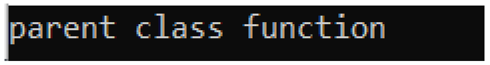
Hierarchical Inheritance in C++ :

- **C++ Multilevel Inheritance:**
- Example:



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

class parent {
  public:
    void Function() {
      cout << "parent class function" ;
    }
};

class Child: public parent {
};

class GrandChild: public Child {
};

int main() {
  GrandChild gc;
  gc.Function();
  return 0;
}
```

Output:

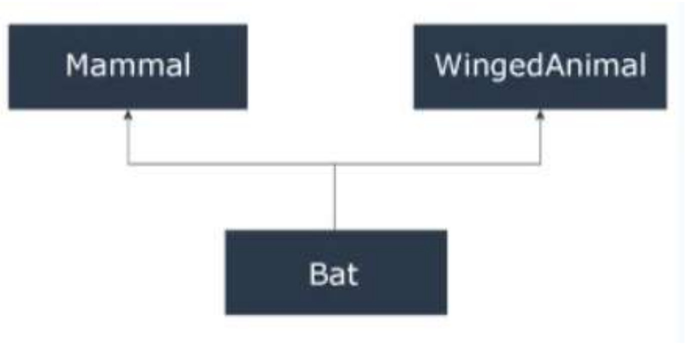


```
parent class function
```

- Example : we have discussed multiple example of the multiple level inheritance,for example:

Multiple inheritance:

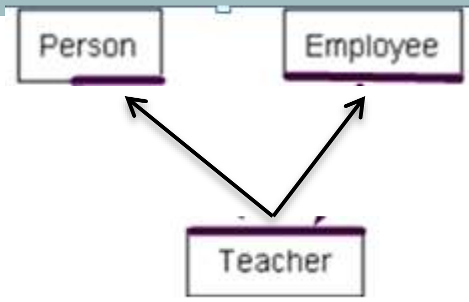
- **Multiple inheritance:**
- In C++ programming, a class can be derived from more than one parent. For example, A class *Bat* is derived from base classes *Mammal* and *WingedAnimal*. It makes sense because bat is a mammal as well as a winged animal.

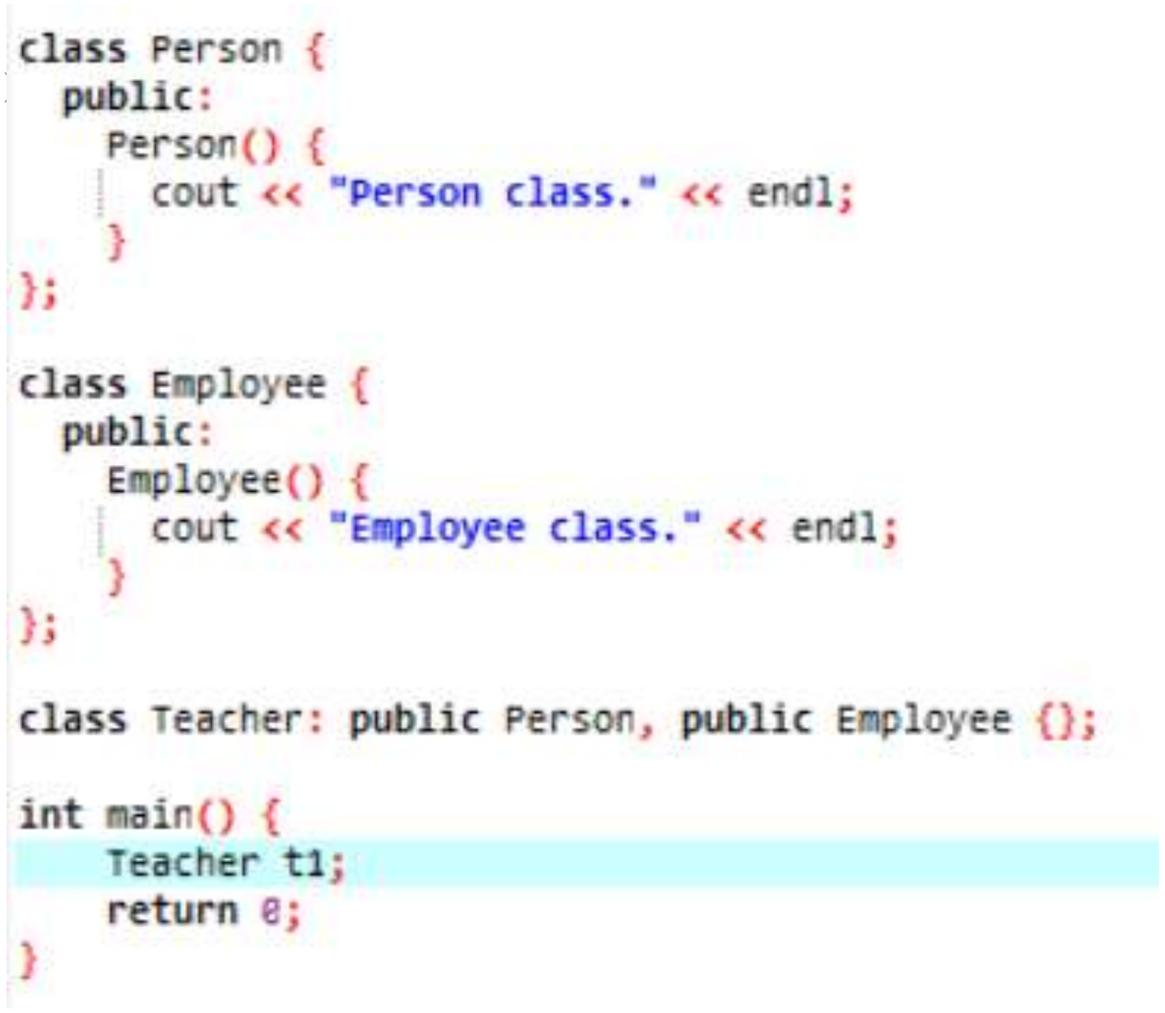


- Teacher employee:
- Person, Employee->Teacher

Multiple inheritance:

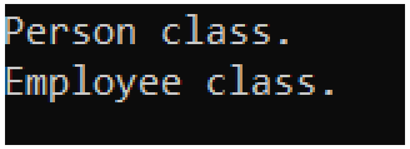
- Teacher employee:





```cpp
class Person {
  public:
    Person() {
      cout << "Person class." << endl;
    }
};

class Employee {
  public:
    Employee() {
      cout << "Employee class." << endl;
    }
};

class Teacher: public Person, public Employee {};

int main() {
    Teacher t1;
    return 0;
}
```

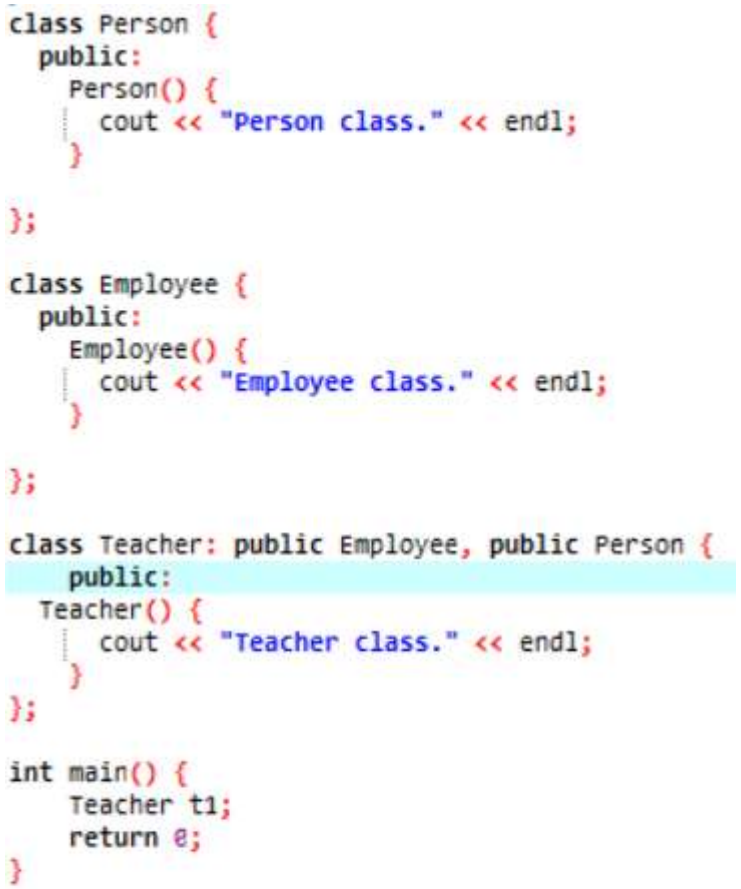
Output:



```
Person class.
Employee class.
```

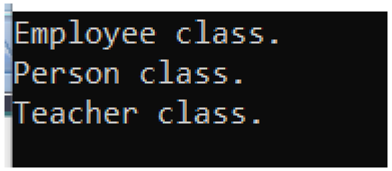
Multiple inheritance:

- The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, Employee's constructor is called before Person's constructor.

- 

```
class Person {
  public:
    Person() {
      cout << "Person class." << endl;
    }

};

class Employee {
  public:
    Employee() {
      cout << "Employee class." << endl;
    }

};

class Teacher: public Employee, public Person {
    public:
  Teacher() {
      cout << "Teacher class." << endl;
    }
};

int main() {
    Teacher t1;
    return 0;
}
```

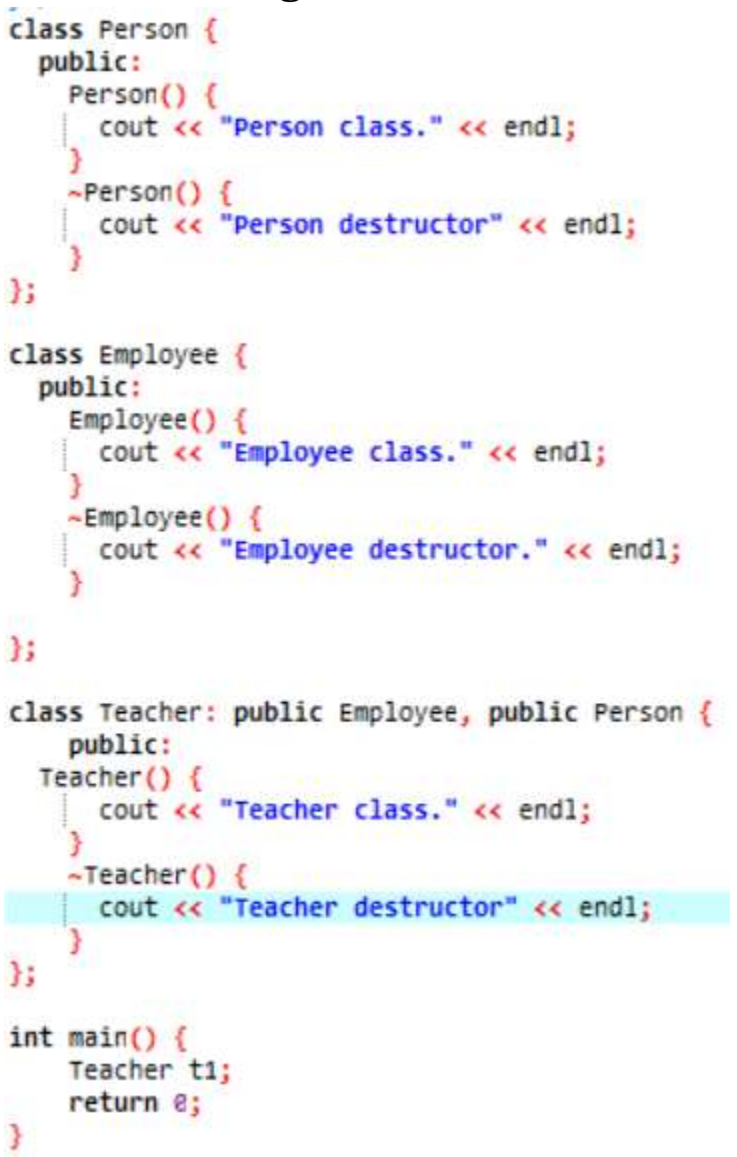
Output:



```
Employee class.
Person class.
Teacher class.
```

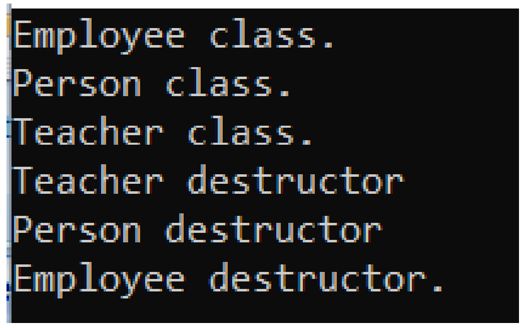
Multiple inheritance:

- Destructor calling.
-



```
class Person {
  public:
    Person() {
      cout << "Person class." << endl;
    }
    ~Person() {
      cout << "Person destructor" << endl;
    }
};

class Employee {
  public:
    Employee() {
      cout << "Employee class." << endl;
    }
    ~Employee() {
      cout << "Employee destructor." << endl;
    }

};

class Teacher: public Employee, public Person {
    public:
  Teacher() {
      cout << "Teacher class." << endl;
    }
    ~Teacher() {
      cout << "Teacher destructor" << endl;
    }
};

int main() {
    Teacher t1;
    return 0;
}
```

Output:



```
Employee class.
Person class.
Teacher class.
Teacher destructor
Person destructor
Employee destructor.
```

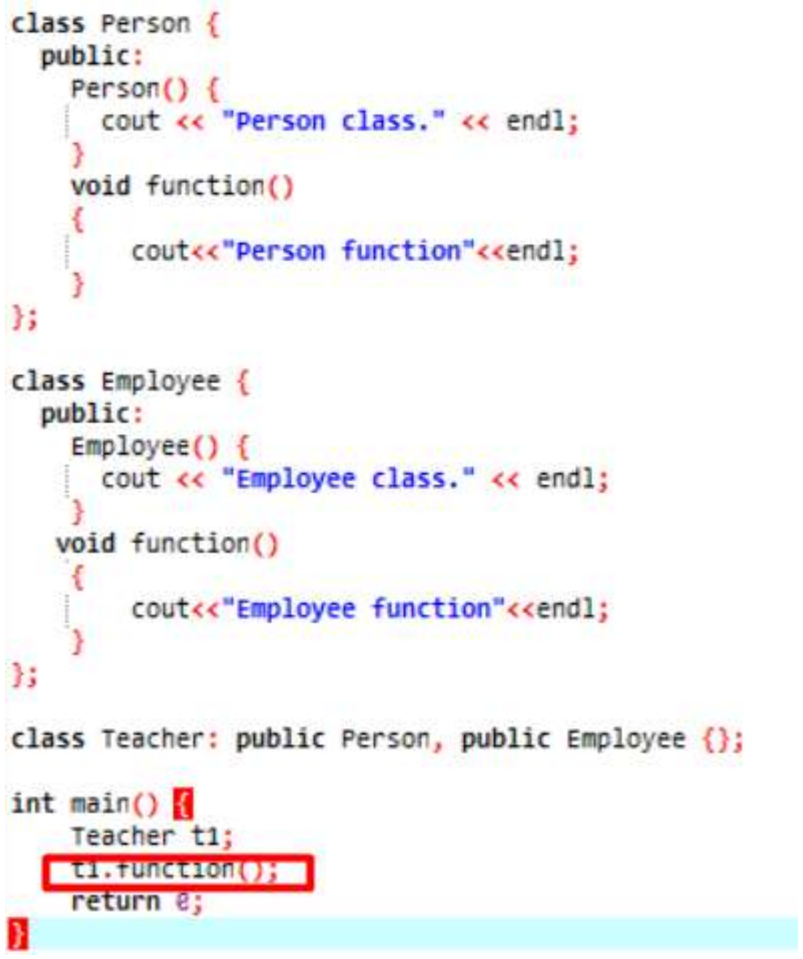
Multiple inheritance:

- **Ambiguity in Multiple Inheritance**
- The most obvious problem with multiple inheritance occurs during function overriding.
- Suppose, two base classes have a same function which is not overridden in derived class.
- If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,

Multiple inheritance:

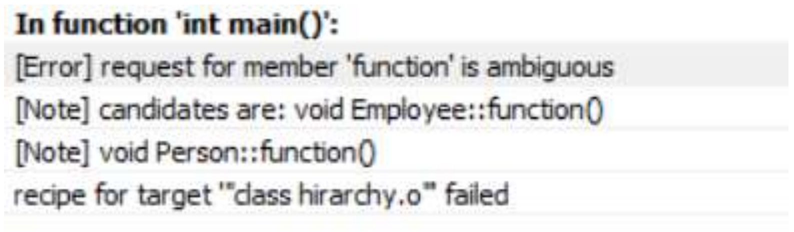
- **Ambiguity in Multiple Inheritance**

- Example:



```
class Person {
  public:
    Person() {
      cout << "Person class." << endl;
    }
    void function()
    {
        cout<<"Person function"<<endl;
    }
};

class Employee {
  public:
    Employee() {
      cout << "Employee class." << endl;
    }
   void function()
    {
        cout<<"Employee function"<<endl;
    }
};

class Teacher: public Person, public Employee {};

int main() {
    Teacher t1;
    t1.function();
    return 0;
}
```

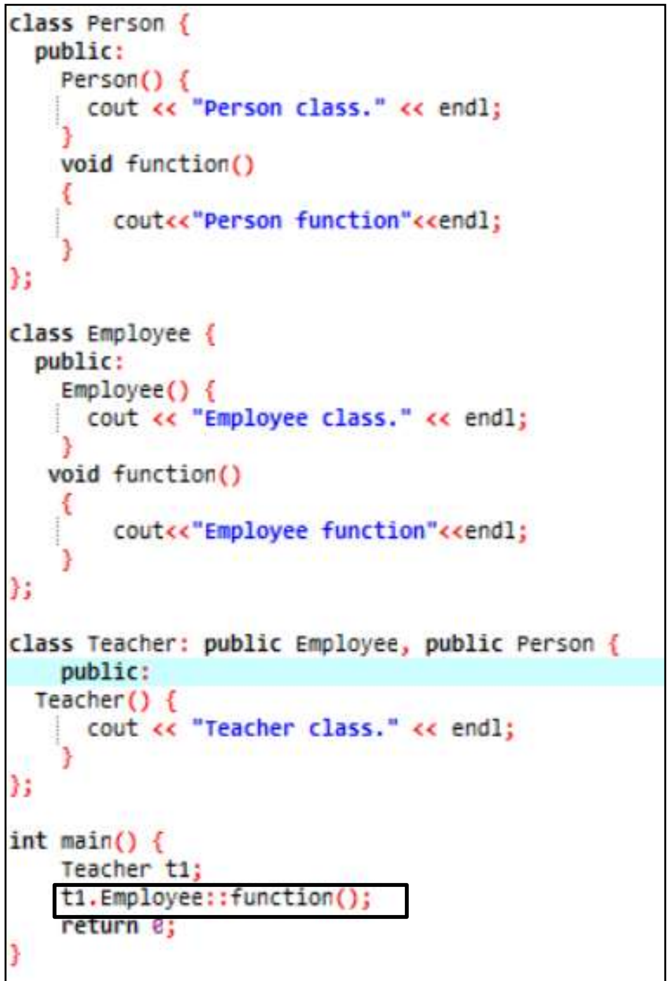
Error:



```
In function 'int main()':
[Error] request for member 'function' is ambiguous
[Note] candidates are: void Employee::function()
[Note] void Person::function()
recipe for target '"class hirarchy.o" failed
```

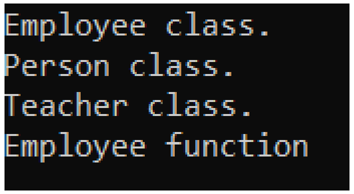
Multiple inheritance:

- **Ambiguity in Multiple Inheritance**
- Therefore, we need to resolve the ambiguous problem in multiple Inheritance. The ambiguity problem can be resolved by defining the class name and scope resolution (::) operator to specify the class from which the member function is invoked in the child class



```
class Person {
  public:
    Person() {
      cout << "Person class." << endl;
    }
    void function()
    {
        cout<<"Person function"<<endl;
    }
};

class Employee {
  public:
    Employee() {
      cout << "Employee class." << endl;
    }
   void function()
    {
        cout<<"Employee function"<<endl;
    }
};

class Teacher: public Employee, public Person {
    public:
  Teacher() {
      cout << "Teacher class." << endl;
    }
};

int main() {
    Teacher t1;
    t1.Employee::function();
    return 0;
}
```

Output:



```
Employee class.
Person class.
Teacher class.
Employee function
```

Credit: Khola Naseem

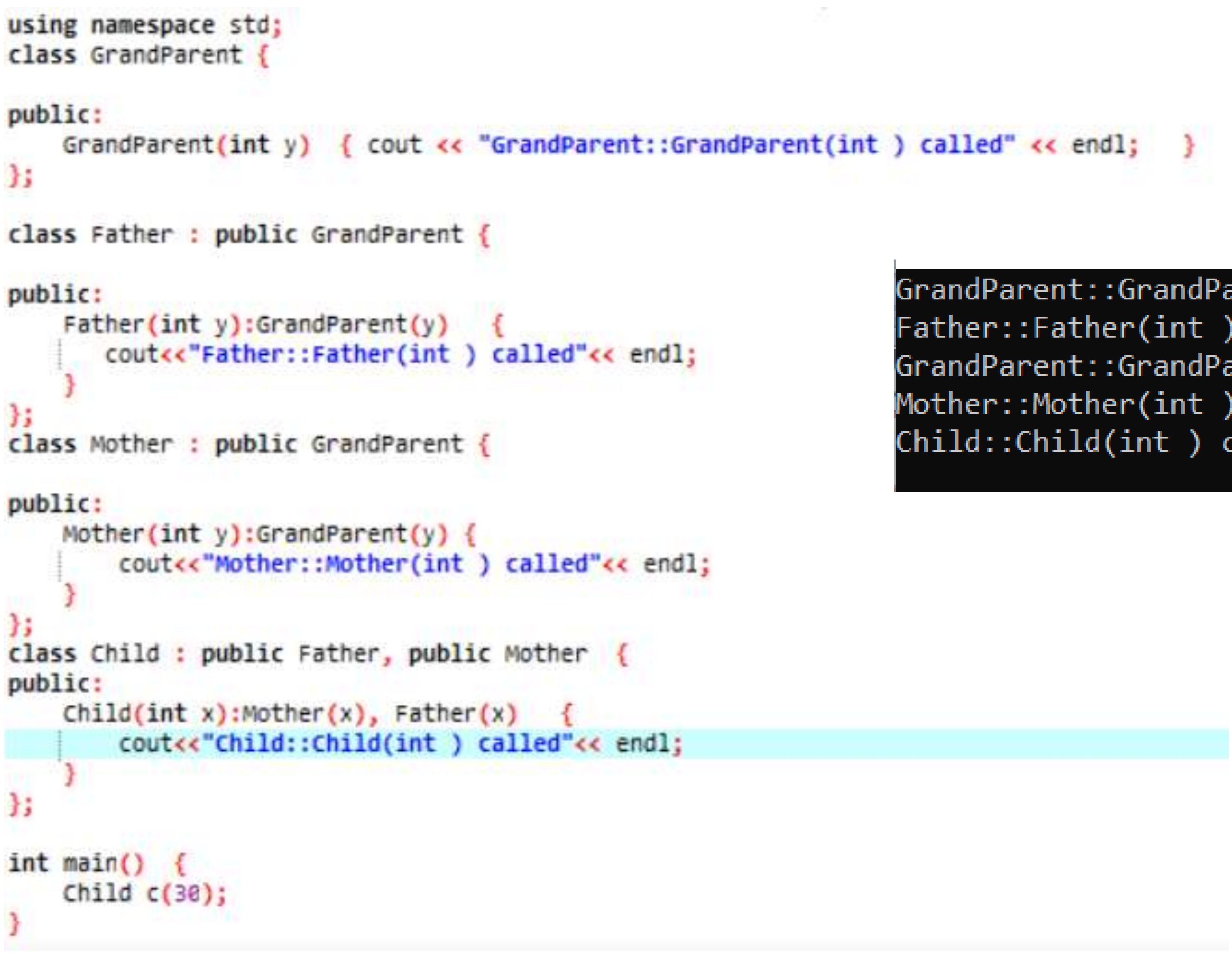
Multiple inheritance:

- **Ambiguity in Multiple Inheritance**
- The diamond problem The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the Child class gets two copies of all attributes of GrandParent class, this causes ambiguities.

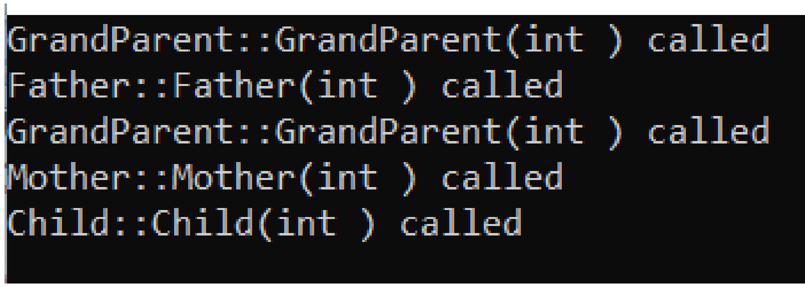
  For example, consider the following program.

Multiple inheritance:

- ➢ **The diamond problem**



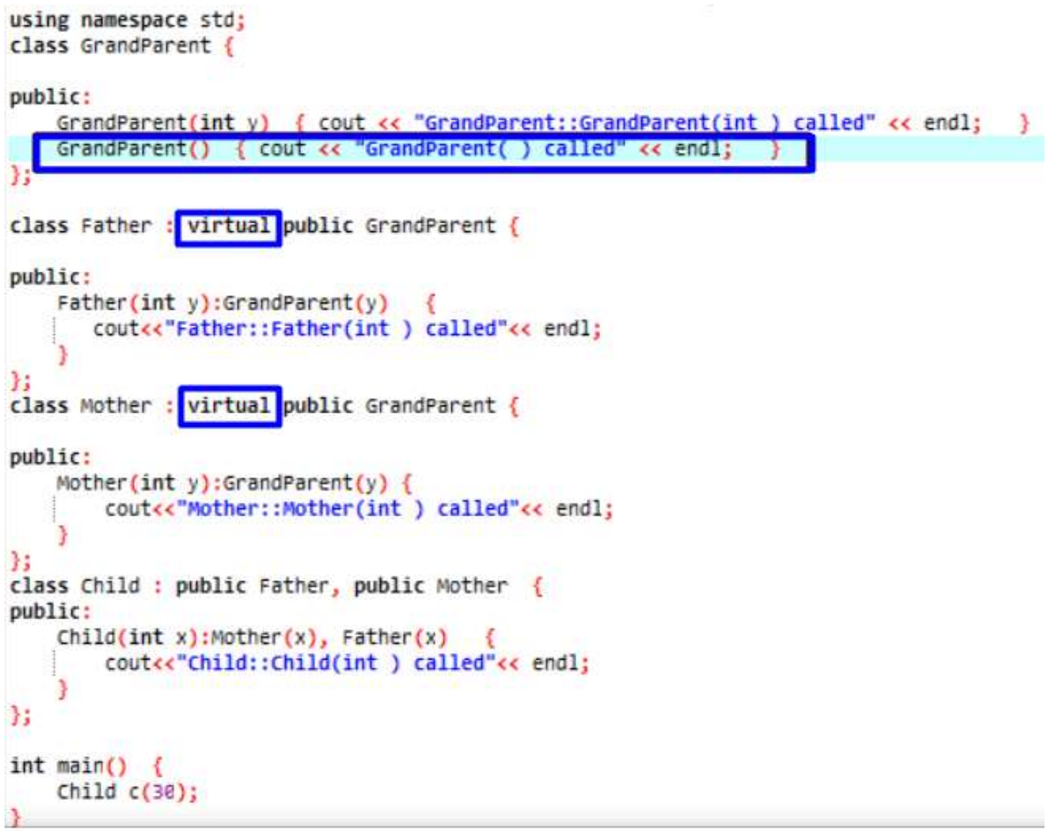
```
using namespace std;
class GrandParent {

public:
    GrandParent(int y)  { cout << "GrandParent::GrandParent(int ) called" << endl;   }
};

class Father : public GrandParent {

public:
    Father(int y):GrandParent(y)   {
        cout<<"Father::Father(int ) called"<< endl;
    }
};
class Mother : public GrandParent {

public:
    Mother(int y):GrandParent(y) {
         cout<<"Mother::Mother(int ) called"<< endl;
    }
};
class Child : public Father, public Mother  {
public:
    Child(int x):Mother(x), Father(x)   {
         cout<<"Child::Child(int ) called"<< endl;
    }
};

int main()  {
    Child c(30);
}
```



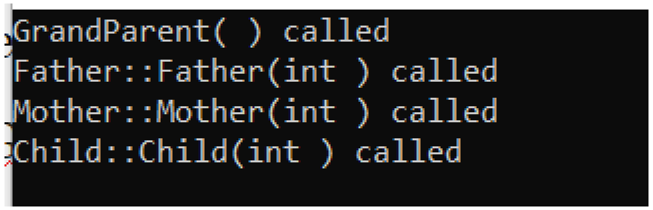
```
GrandParent::GrandParent(int ) called
Father::Father(int ) called
GrandParent::GrandParent(int ) called
Mother::Mother(int ) called
Child::Child(int ) called
```

Multiple inheritance:

- **Solution to diamond problem**
- In the above program, constructor of 'GrandParent' is called two times. Destructor of 'GrandParent' will also be called two times when object 'c' is destructed.
- So object 'c' has two copies of all members of 'GrandParent', this causes ambiguities. The solution to this problem is '**virtual**' keyword. We make the classes 'Father' and 'Mother' as virtual base classes to avoid two copies of 'GrandParent' in 'Child' class.



```
using namespace std;
class GrandParent {

public:
    GrandParent(int y)  { cout << "GrandParent::GrandParent(int ) called" << endl;   }
    GrandParent()  { cout << "GrandParent( ) called" << endl;   }
};

class Father : virtual public GrandParent {

public:
    Father(int y):GrandParent(y)   {
       cout<<"Father::Father(int ) called"<< endl;
    }
};
class Mother : virtual public GrandParent {

public:
    Mother(int y):GrandParent(y) {
        cout<<"Mother::Mother(int ) called"<< endl;
    }
};
class Child : public Father, public Mother  {
public:
    Child(int x):Mother(x), Father(x)   {
        cout<<"Child::Child(int ) called"<< endl;
    }
};

int main()  {
    Child c(30);
}
```

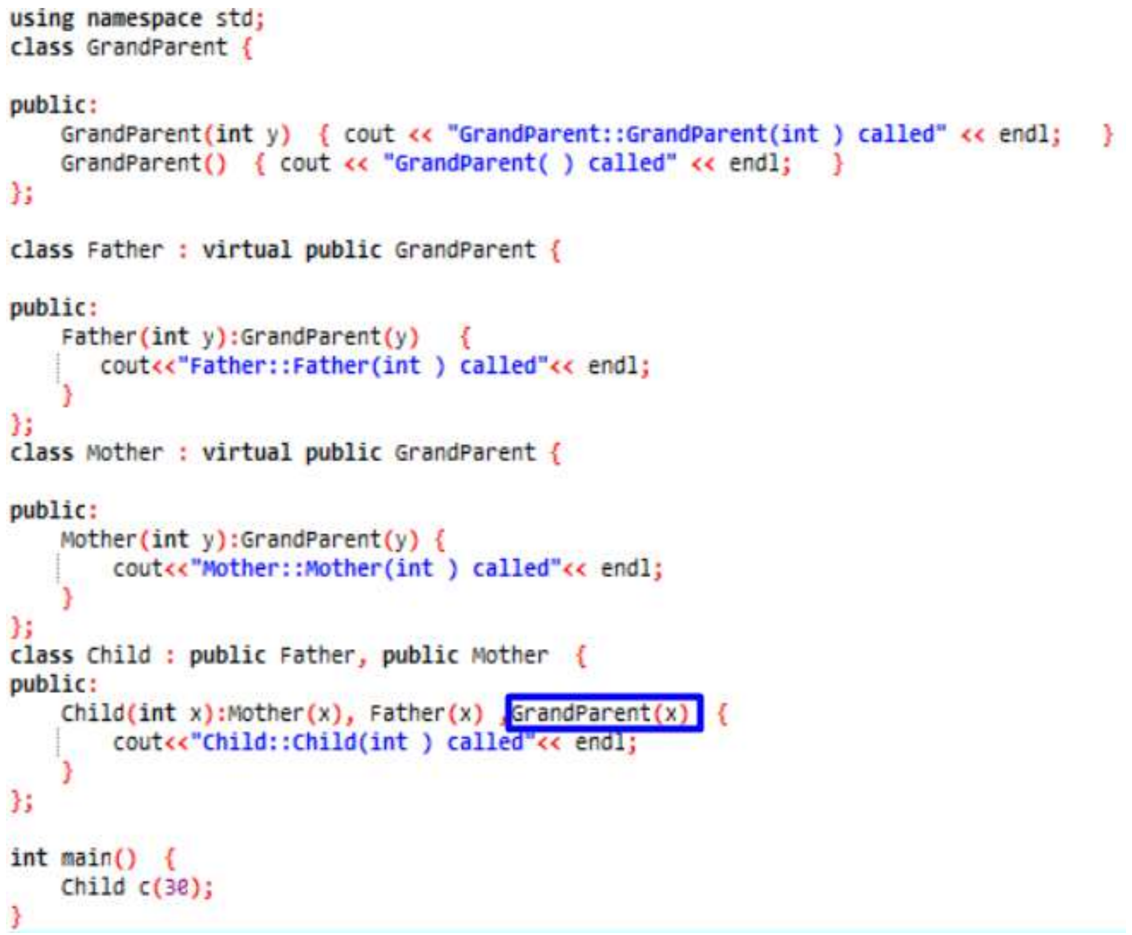


```
GrandParent( ) called
Father::Father(int ) called
Mother::Mother(int ) called
Child::Child(int ) called
```

Credit: Khola Naseem

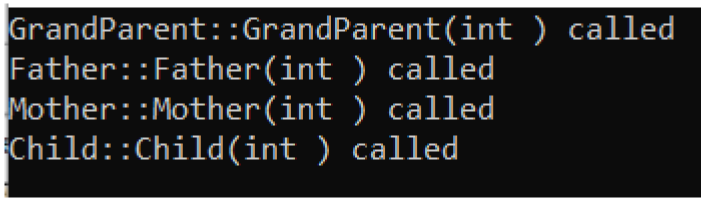
Multiple inheritance:

- **Solution to diamond problem**
- In the above program, constructor of ‘GrandParent’ is called two times. Destructor of ‘GrandParent’ will also be called two times when object ‘c’ is destructed.
- So object ‘c’ has two copies of all members of ‘GrandParent’, this causes ambiguities. The solution to this problem is ‘**virtual**’ keyword. We make the classes ‘Father’ and ‘Mother’ as virtual base classes to avoid two copies of ‘GrandParent’ in ‘Child’ class.



```
using namespace std;
class GrandParent {

public:
    GrandParent(int y)  { cout << "GrandParent::GrandParent(int ) called" << endl;   }
    GrandParent()  { cout << "GrandParent( ) called" << endl;   }
};

class Father : virtual public GrandParent {

public:
    Father(int y):GrandParent(y)   {
        cout<<"Father::Father(int ) called"<< endl;
    }
};
class Mother : virtual public GrandParent {

public:
    Mother(int y):GrandParent(y) {
        cout<<"Mother::Mother(int ) called"<< endl;
    }
};
class Child : public Father, public Mother  {
public:
    Child(int x):Mother(x), Father(x) ,GrandParent(x) {
        cout<<"Child::Child(int ) called"<< endl;
    }
};

int main()  {
    Child c(30);
}
```

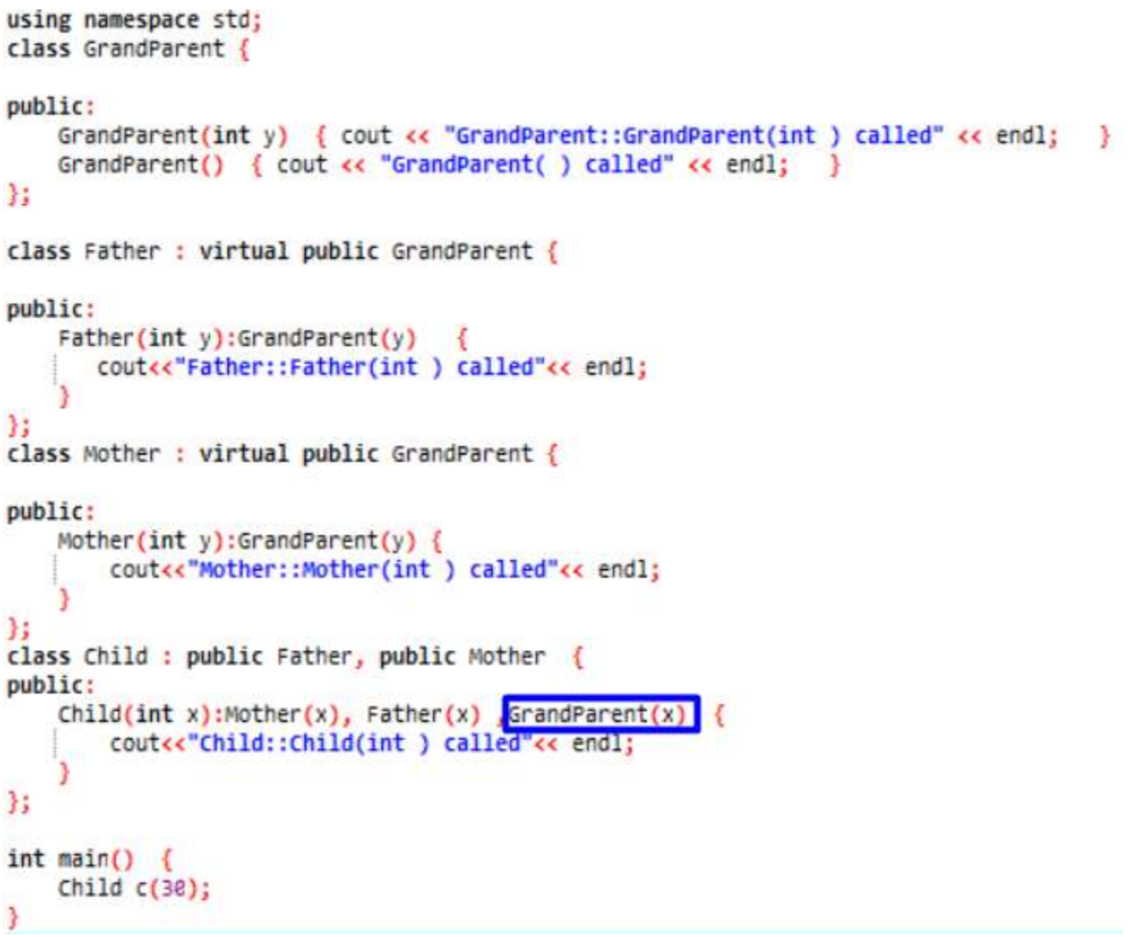
Output:



```
GrandParent::GrandParent(int ) called
Father::Father(int ) called
Mother::Mother(int ) called
Child::Child(int ) called
```

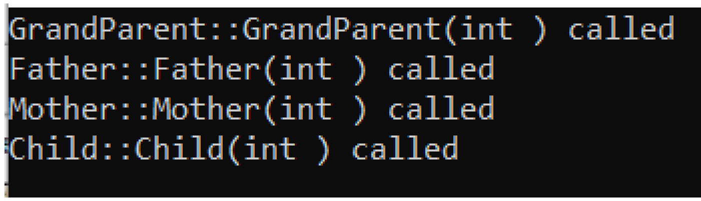
Multiple inheritance:

- **Solution to diamond problem**
- In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class. It is allowed only when 'virtual' keyword is used.
- As an exercise, predict the output of following programs.



```
using namespace std;
class GrandParent {

public:
    GrandParent(int y)  { cout << "GrandParent::GrandParent(int ) called" << endl;   }
    GrandParent()  { cout << "GrandParent( ) called" << endl;   }
};

class Father : virtual public GrandParent {

public:
    Father(int y):GrandParent(y)   {
        cout<<"Father::Father(int ) called"<< endl;
    }
};
class Mother : virtual public GrandParent {

public:
    Mother(int y):GrandParent(y) {
        cout<<"Mother::Mother(int ) called"<< endl;
    }
};
class Child : public Father, public Mother  {
public:
    Child(int x):Mother(x), Father(x) ,GrandParent(x) {
        cout<<"Child::Child(int ) called"<< endl;
    }
};

int main()  {
    Child c(30);
}
```

Output:



```
GrandParent::GrandParent(int ) called
Father::Father(int ) called
Mother::Mother(int ) called
Child::Child(int ) called
```

Compile time Polymorphism

- C++ programming language has a mechanism known as compile time polymorphism, also referred to as static polymorphism or early binding, that enables the choice of the appropriate method or function to be made at compile time.
- The compile time polymorphism in C++ is a type of polymorphism, which refers to the ability of a programming language to determine the appropriate method or function to call at compile time-based on the types of arguments being passed
- There are a couple of ways to achieve compile time polymorphism:
  - Function Overloading
  - Operator Overloading

Compile time Polymorphism:

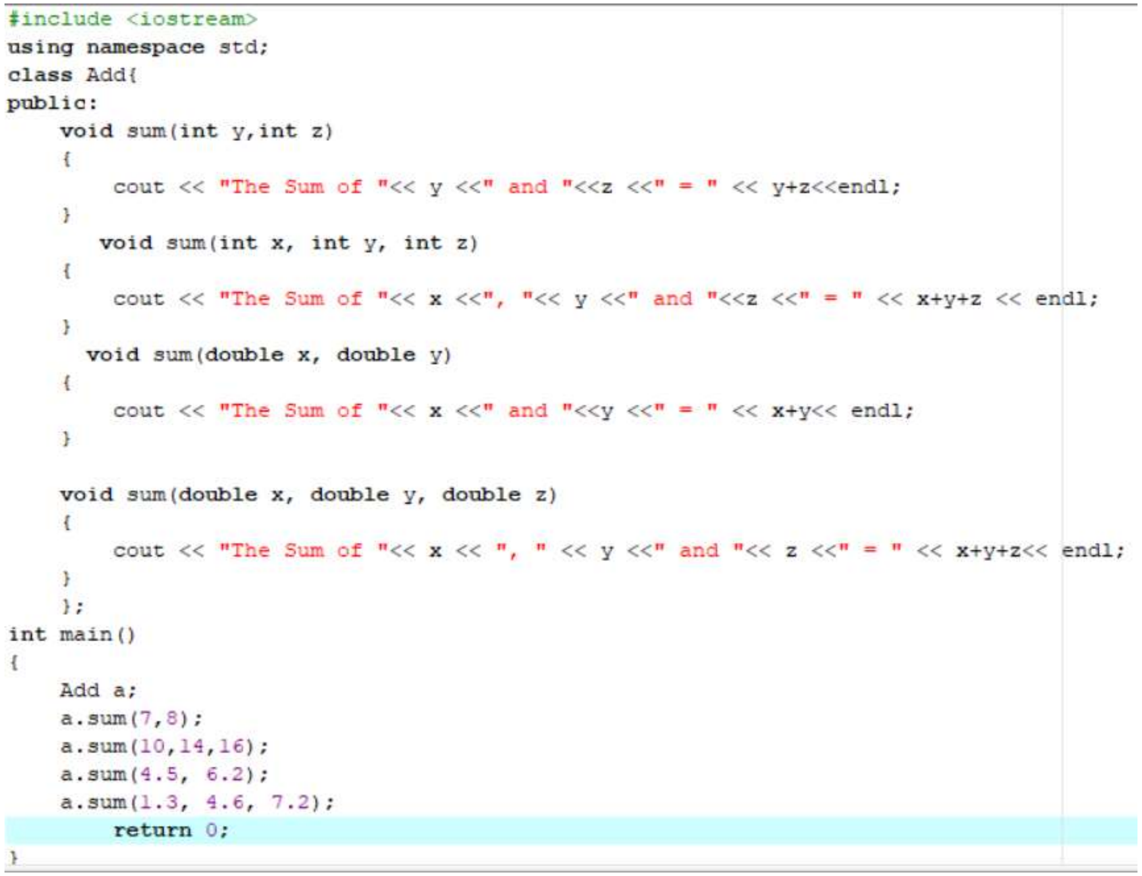
- There are a couple of ways to achieve compile time polymorphism:
  - Function Overloading
  - Operator Overloading

Compile time Polymorphism:

- Function Overloading
  - C++ language's function overloading feature enables us to define multiple functions that have the same name but different parameters.
  - When we want to perform the same operation on various data types, or when we want to offer different levels of functionality depending on the quantity or kind of arguments passed, this can be helpful.
  - The compiler chooses which version of the function to call based on the number, type, and ordering of the arguments passed.

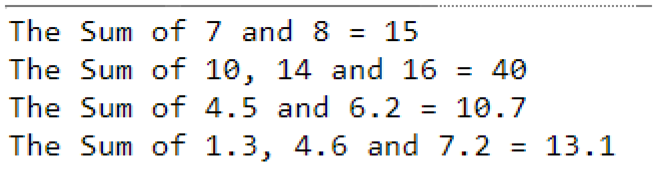
Compile time Polymorphism:

- Function Overloading



```
#include <iostream>
using namespace std;
class Add{
public:
    void sum(int y,int z)
    {
        cout << "The Sum of "<< y <<" and "<<z <<" = " << y+z<<endl;
    }
       void sum(int x, int y, int z)
    {
        cout << "The Sum of "<< x <<", "<< y <<" and "<<z <<" = " << x+y+z << endl;
    }
      void sum(double x, double y)
    {
        cout << "The Sum of "<< x <<" and "<<y <<" = " << x+y<< endl;
    }

    void sum(double x, double y, double z)
    {
        cout << "The Sum of "<< x << ", " << y <<" and "<< z <<" = " << x+y+z<< endl;
    }
    };
int main()
{
    Add a;
    a.sum(7,8);
    a.sum(10,14,16);
    a.sum(4.5, 6.2);
    a.sum(1.3, 4.6, 7.2);
        return 0;
}
```

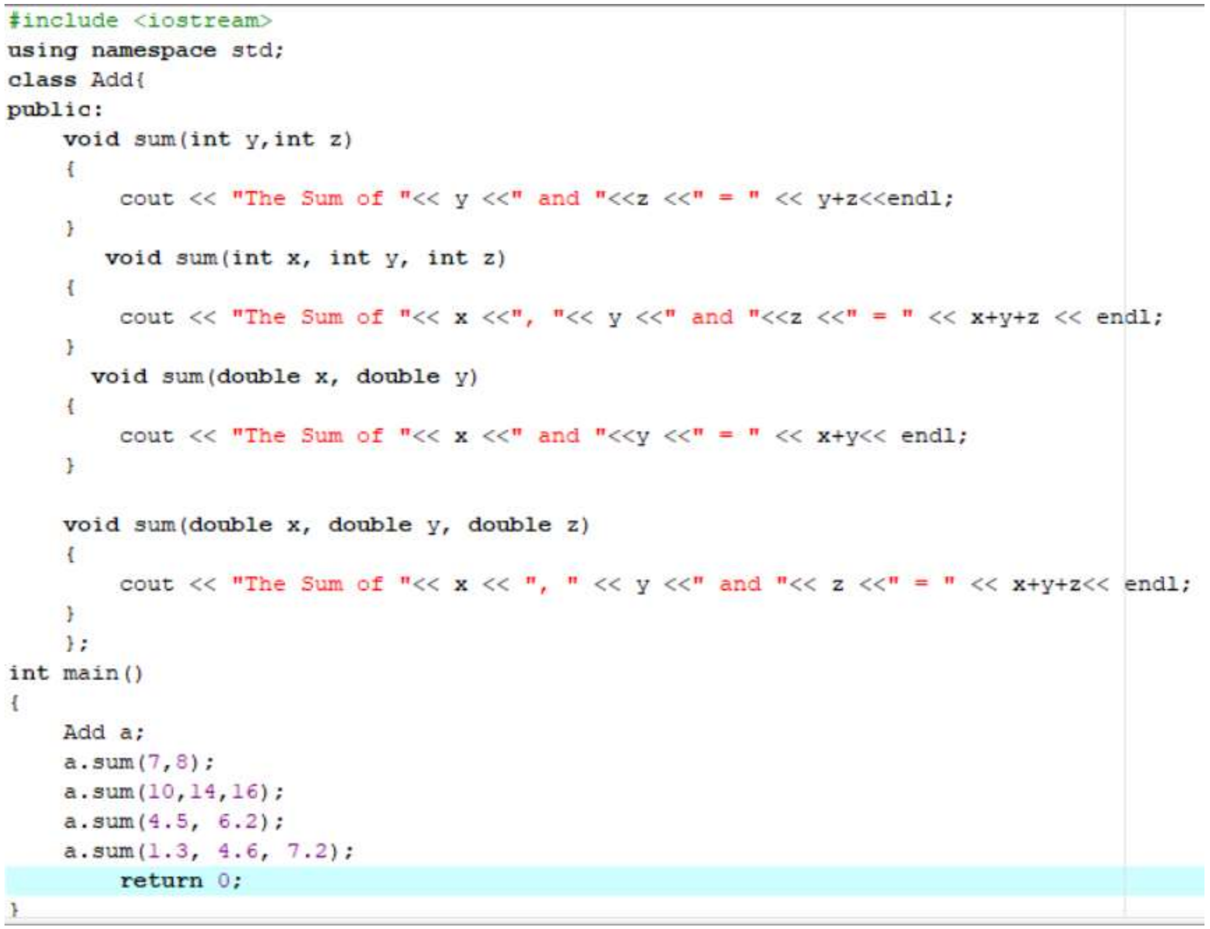
Output:



```
The Sum of 7 and 8 = 15
The Sum of 10, 14 and 16 = 40
The Sum of 4.5 and 6.2 = 10.7
The Sum of 1.3, 4.6 and 7.2 = 13.1
```

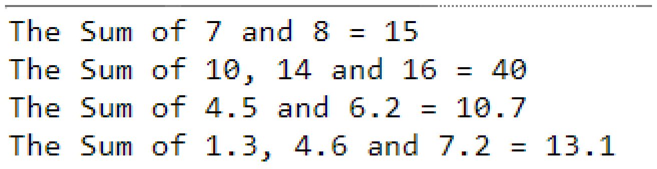
Compile time Polymorphism:

➢ Function Overloading



```
#include <iostream>
using namespace std;
class Add{
public:
    void sum(int y,int z)
    {
        cout << "The Sum of "<< y <<" and "<<z <<" = " << y+z<<endl;
    }
       void sum(int x, int y, int z)
    {
        cout << "The Sum of "<< x <<", "<< y <<" and "<<z <<" = " << x+y+z << endl;
    }
      void sum(double x, double y)
    {
        cout << "The Sum of "<< x <<" and "<<y <<" = " << x+y<< endl;
    }

    void sum(double x, double y, double z)
    {
        cout << "The Sum of "<< x << ", " << y <<" and "<< z <<" = " << x+y+z<< endl;
    }
    };
int main()
{
    Add a;
    a.sum(7,8);
    a.sum(10,14,16);
    a.sum(4.5, 6.2);
    a.sum(1.3, 4.6, 7.2);
        return 0;
}
```

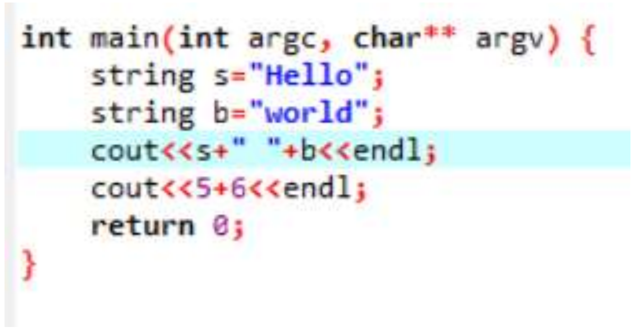
Output:



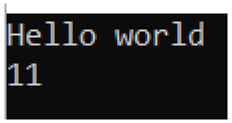
```
The Sum of 7 and 8 = 15
The Sum of 10, 14 and 16 = 40
The Sum of 4.5 and 6.2 = 10.7
The Sum of 1.3, 4.6 and 7.2 = 13.1
```

Compile time Polymorphism:

- Operator overloading:
    - C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
    - For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands.
    - So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.



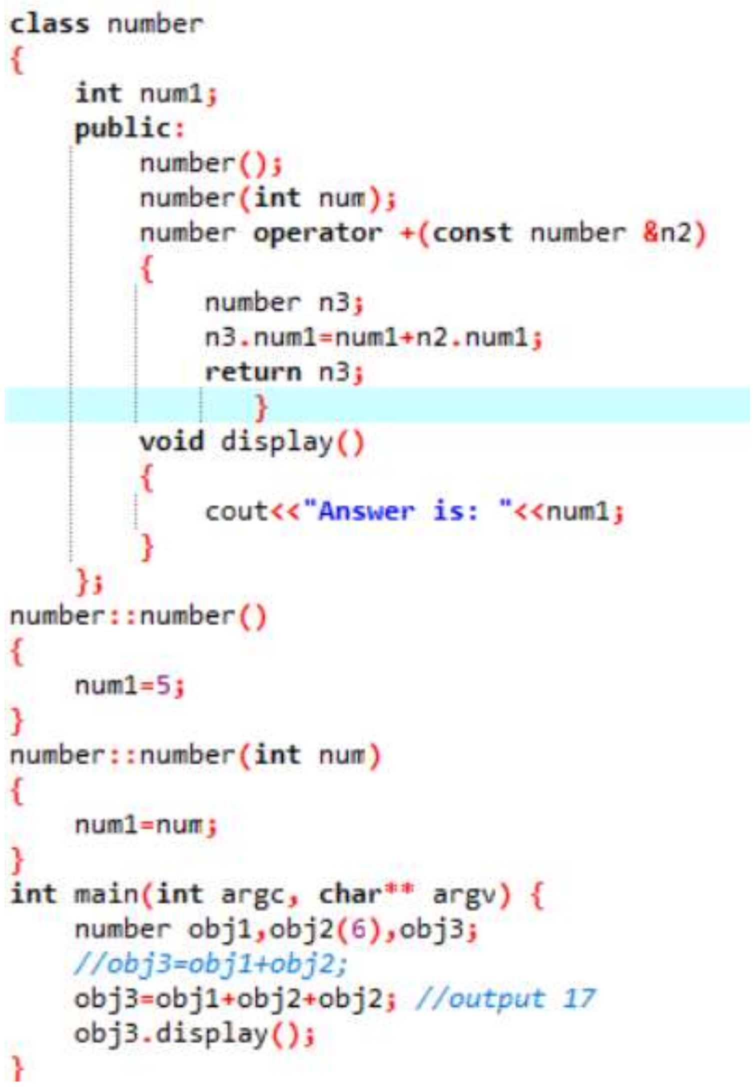
```
int main(int argc, char** argv) {
    string s="Hello";
    string b="world";
    cout<<s+" "+b<<endl;
    cout<<5+6<<endl;
    return 0;
}
```



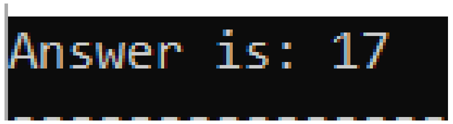
```
Hello world
11
```

Compile time Polymorphism:

- Operator overloading:
  - Multiple examples are discussed in the class.



```
class number
{
    int num1;
    public:
        number();
        number(int num);
        number operator +(const number &n2)
        {
            number n3;
            n3.num1=num1+n2.num1;
            return n3;
        }
        void display()
        {
            cout<<"Answer is: "<<num1;
        }
    };
number::number()
{
    num1=5;
}
number::number(int num)
{
    num1=num;
}
int main(int argc, char** argv) {
    number obj1,obj2(6),obj3;
    //obj3=obj1+obj2;
    obj3=obj1+obj2+obj2; //output 17
    obj3.display();
}
```



```
Answer is: 17
```

Credit: Khola Naseem

Polymorphism Compilation advantage:

- Compile-time polymorphism can help you write cleaner, more efficient code by allowing you to reuse function names and reduce redundancy.
- It also helps catch errors at compile time instead of at runtime, which can save you time and effort in debugging.

Reference material

- **For Practice Questions, refer to these books**
  - C++ Programming From Problem Analysis To Program Design, 5th Edition, D.S.Malik. Chapter 12.
  - C++ How to Program, Deitel & Deitel, 5th Edition, Prentice Hall.
  - Object Oriented Programming in C++ by Robert Lafore.
  - Object Oriented Software Construction, Bertrand Meyer's
  - Object-Oriented Analysis and Design with applications, Grady Booch et al, 3Rd Edition, Pearson, 2007
  - Web