

Inheritance

What is Inheritance

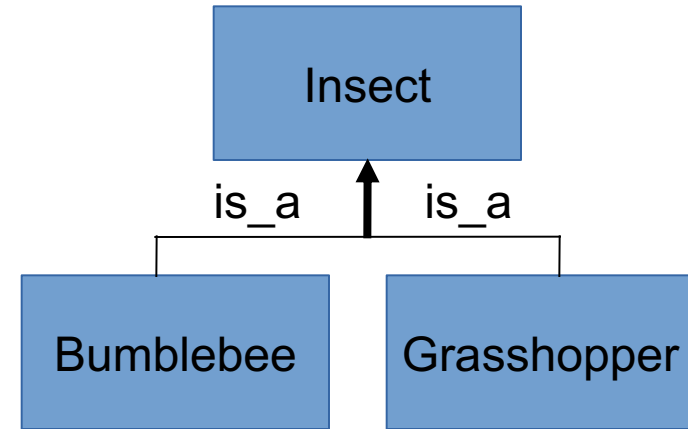
Inheritance is a way of creating a new class by starting with an existing class and adding new members

The new class is a specialized version of the existing class

The new class can replace or extend the functionality of the existing class

Inheritance establishes an "is a" relationship between classes.

- A poodle is a dog
- A car is a vehicle
- A flower is a plant
- A football player is an athlete



UML representation

Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	FacultyMember, StaffMember
Account	CheckingAccount, SavingsAccount

Inheritance

Base class (Super class or parent) – inherited from

Derived class (Sub class or child) – inherits from the base class

Notation:

```
class Student  
{  
  
  
  
  
  
  
};
```

```
class UnderGrad : public student  
{  
  
  
};
```

Derived Class
Sub Class
Child Class

Base Class
Super Class
Parent Class

Shapes

```
class Shape {
private:
    int x;
    int y;
public:
    Shape() {
        cout << "def Shape ..." << endl;
        x = y = 0;
    }
    Shape(int x, int y) {
        cout << "param Shape ..." << endl;
        this->x = x;
        this->y = y;
    }
    void print () {
        cout << "print Shape fun..." << endl;
        cout << "(" << x << ", "
            << y << ")" << endl;
    }
    int getX(){ return x;}
    int getY(){ return y;}
    void setX(int v){ x=v;}
    void setY(int v){ y=v;}
};
```

```
class Rectangle: public Shape {
private:
    int width;
    int height;
public:
    Rectangle() {
        cout << "def. Rectangle ..." << endl;
        this->setX(0);
        this->setY(0);
        width = height = 0;
    }
    Rectangle(int x, int y, int w, int h) {
        cout << "Param. Rectangle ..." << endl;
        this->setX(x);
        this->setY(y);
        width = w;
        height = h;
    }
    void print() {
        cout << "Rectangle print fun ..." << endl;
        cout << "Rectangle at" << endl;
        cout << getX() << ", " << getY() << endl;
        cout << width << ", " << height << endl;
    }
};
```

Shapes

```
int main( ) {  
    cout << " ----1----" << endl;  
    Shape s(0,1);  
  
    cout << " ----2----" << endl;  
    Rectangle r(1,2,5,6);  
  
    cout << " ----3----" << endl;  
    s.print();  
  
    cout << " ----4----" << endl;  
    r.print();  
  
    cout << " ----5----" << endl;  
    return 0;  
}
```

Output

```
----1----  
param Shape ...  
----2----  
def Shape ...  
Param. Rectangle ...  
----3----  
print Shape fun...  
(0,1)  
----4----  
Rectangle print fun ...  
Rectangle at  
1,2  
5,6  
----5----
```

Shape

Rectangle

Objects of derived classes

An object of a derived class 'is a(n)'
object of the base class

an UnderGrad is a Student
a Mammal is an Animal

- ✗ A derived object has all of the characteristics of the base class
- ✗ An object of the derived class has:
 - ✗ all members defined in child class
 - ✗ all members declared in parent class
- ✗ An object of the derived class can use:
 - ✗ all public members defined in child class
 - ✗ all public members defined in parent class

```
class Shape {  
private:  
    int x;  
    int y;  
public:  
    Shape();  
    Shape(int x, int y);  
    void print ();  
    int getX();  
    int getY()  
};
```

```
class Rectangle: public Shape {  
private:  
    int width;  
    int height;  
public:  
    Rectangle();  
    Rectangle(int x, int y, int w, int h);  
    void print();  
};
```

- **Rectangle class can access all public and protected members of Shape**

Protected Members and Class Access

Protected member access specification: like **private**, but accessible by objects of derived class.

Class access specification: determines how **private**, **protected**, and **public** members of base class are inherited by the derived class

- 1) **public** – object of derived class can be treated as object of base class (not vice-versa)
- 2) **protected** – more restrictive than `public`, but allows derived classes to know details of parents
- 3) **private** – prevents objects of derived class from being treated as objects of base class.

Inheritance vs Access

Base class members

```
private: x  
protected: y  
public: z
```

private
base class

How inherited base class
members
appear in derived class

```
x is inaccessible  
private: y  
private: z
```

```
private: x  
protected: y  
public: z
```

protected
base class

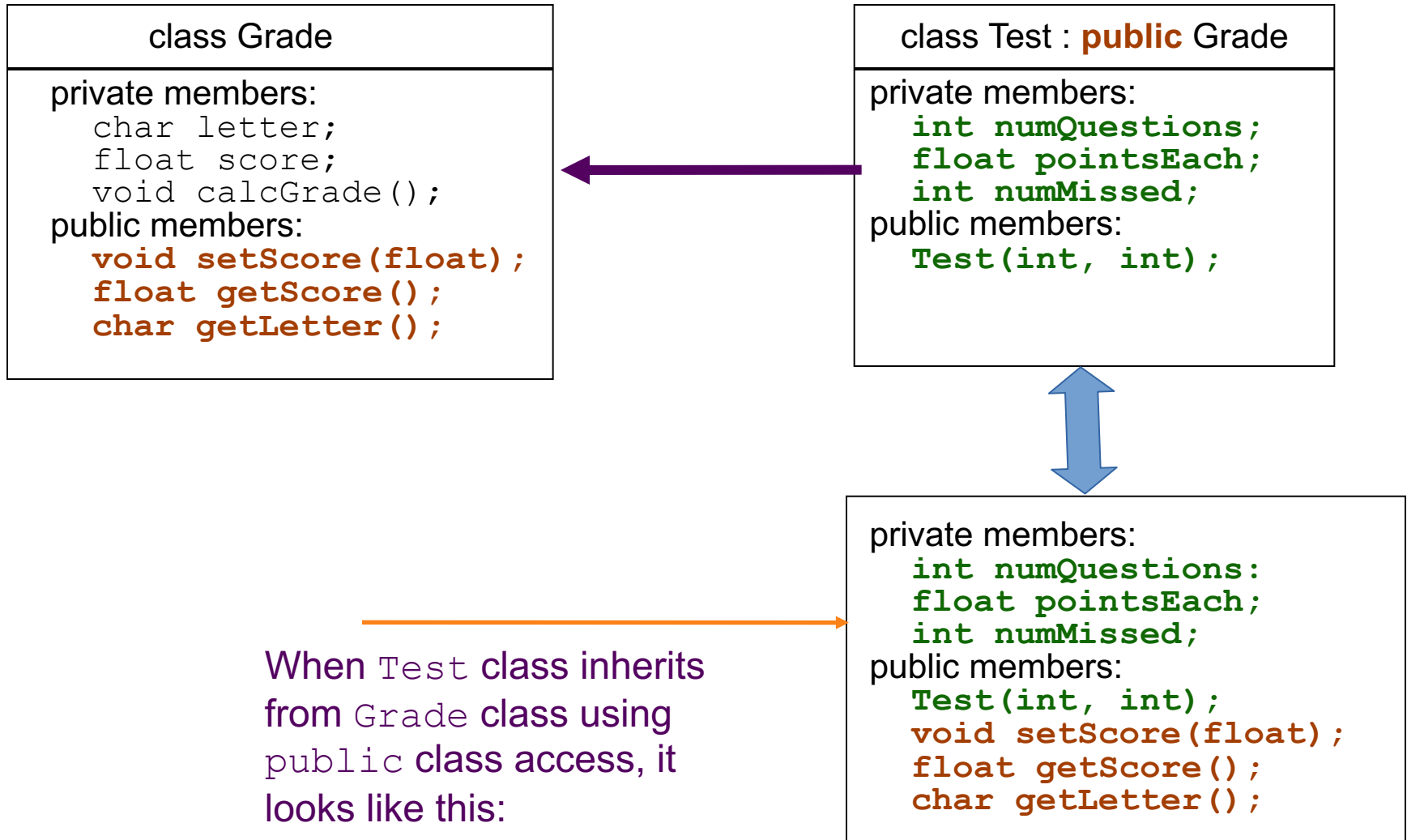
```
x is inaccessible  
protected: y  
protected: z
```

```
private: x  
protected: y  
public: z
```

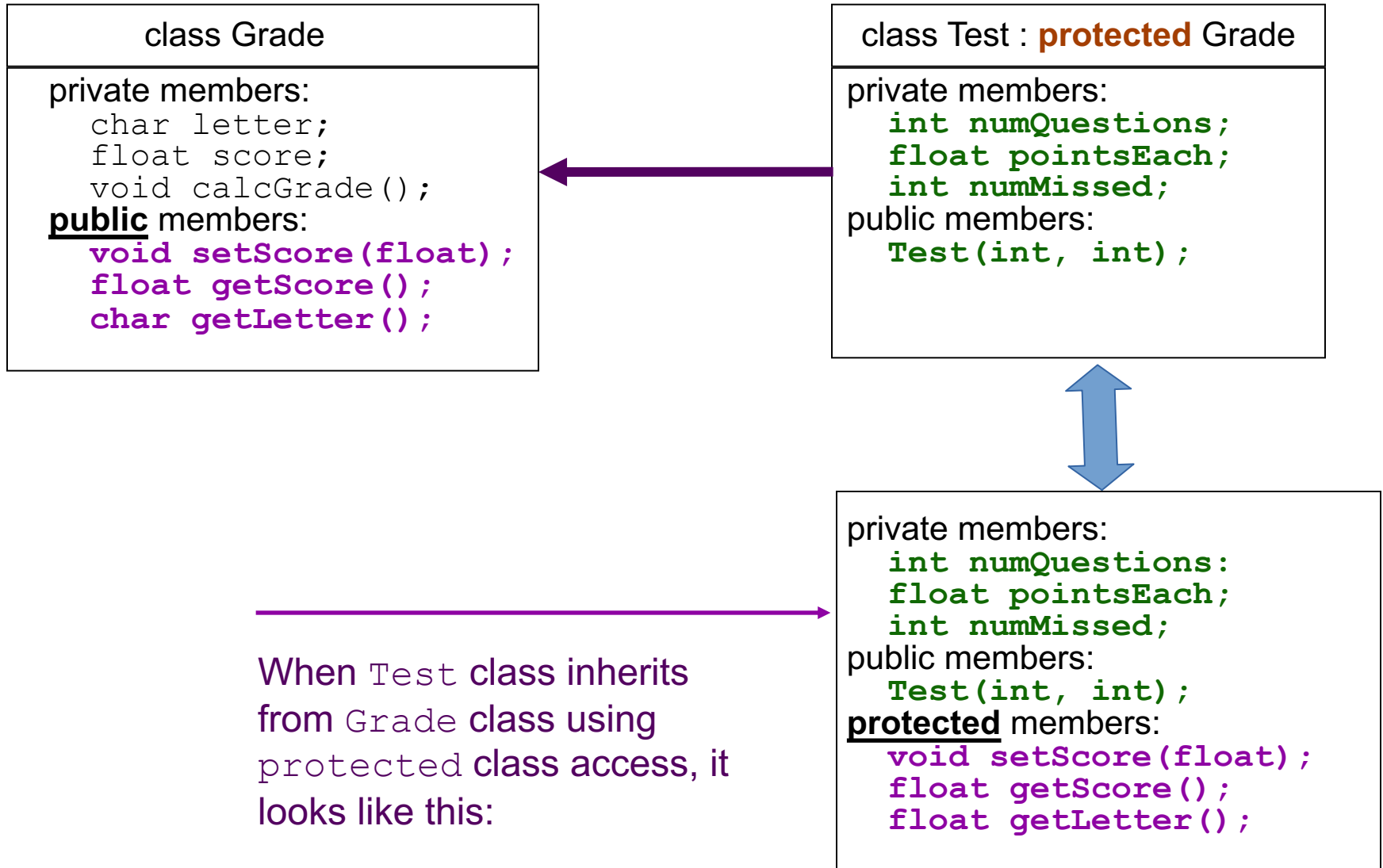
public
base class

```
x is inaccessible  
protected: y  
public: z
```

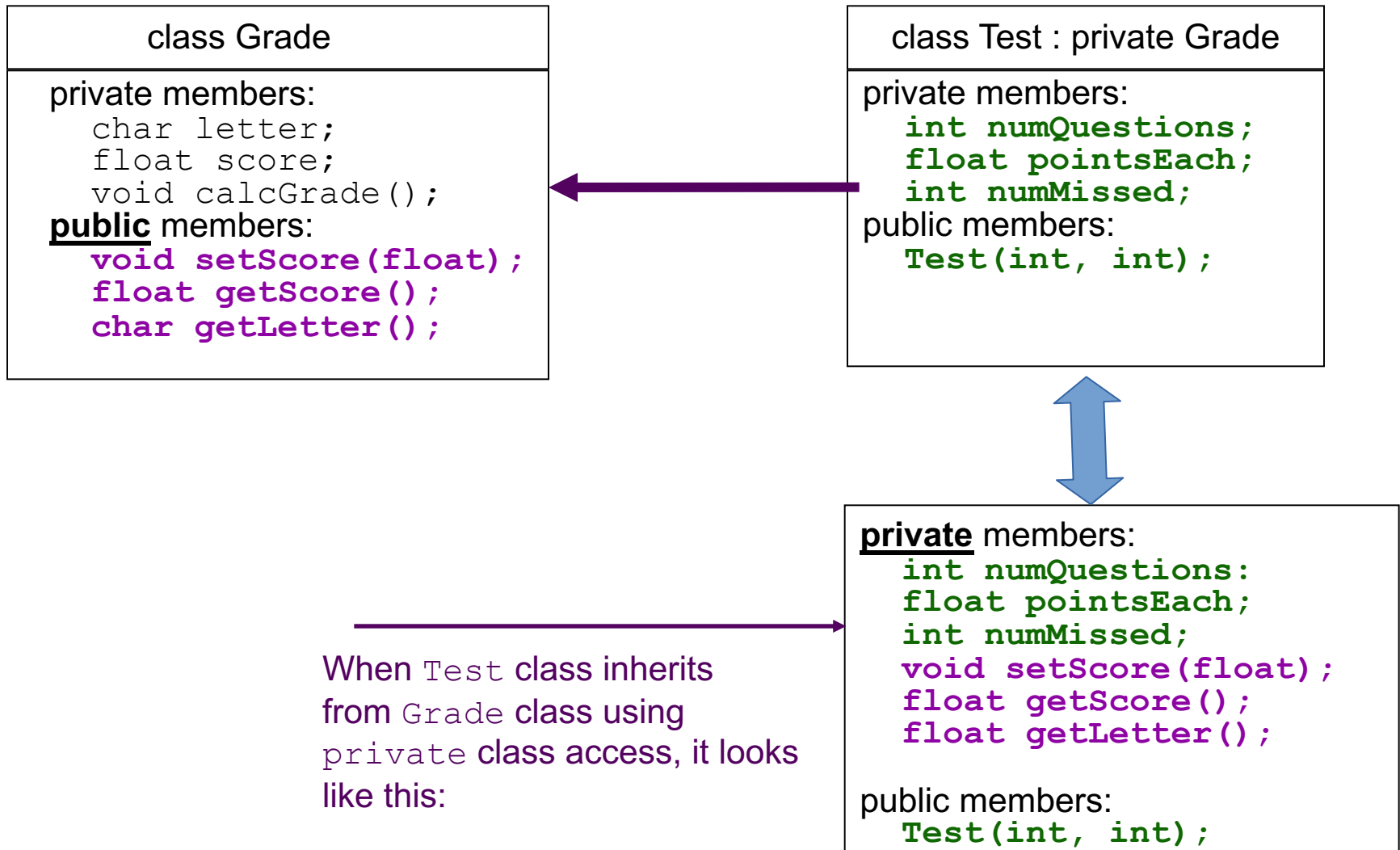

Public Access



Protected Access



Private Access



Constructors and Destructors

Derived classes can have their own constructors and destructors

When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor

When an object of a derived class is destroyed, its destructor is called first, then that of the base class

```
class Parent{  
public:  
    Parent(){  
        cout << "Parent()" << endl;  
    }  
    ~Parent(){  
        cout << "~Parent" << endl;  
    }  
};
```

```
class Child: public Parent{  
public:  
    Child(){  
        cout << "Child()" << endl;  
    }  
    ~Child(){  
        cout << "~Child" << endl;  
    }  
};
```

```
int main()  
{  
    Parent p;  
  
    return 0;  
}
```

```
Parent()  
~Parent
```

```
int main()  
{  
    Child c;  
  
    return 0;  
}
```

```
Parent()  
Child()  
~Child  
~Parent
```

Constructors and Destructors

```
class Shape {  
private:  
    int x;  
    int y;  
    string label;  
public:  
    Shape();  
    Shape(int,int);  
    Shape(int,int, string);  
    void print();  
    void setX(int);  
    void setY(int);  
    int getX();  
    int getY();  
    ~Shape();  
};
```

is_a

```
class Circle:public Shape {  
private:  
    double radius;  
    string label;  
public:  
    Circle();  
    Circle(int,int);  
    Circle(int, int, double, const string&);  
    void print();  
    void setCenter(int,int);  
    void setRadius(double r);  
    double getRadius();  
    ~Circle();  
};
```

Constructors and Destructors

```
Shape::Shape() {
    x = y = 0;
    label = "Shape";
    cout << label << " is created ()\n";
}
Shape::Shape(int x, int y){
    this->x = x;  this->y = y;
    label = "Shape";
    cout << label
        << " is created (int,int)\n";
}
Shape::Shape(int x, int y, string
label){
    cout << label
        << " is created (int,int,string)\n";
    this->x = x;  this->y = y;
    this->label = label;
}
void Shape::print(){
    cout << label << ".print ->";
    cout << "(" << x << ", " << y << ")\n";
}
Shape::~Shape() {
    cout << label << " is destroyed \n";
}
```

```
Circle::Circle() {
    radius = 1.0;
    label = "C";
    cout << "Circle: " << label
        << " is created()\n";
}
Circle::Circle(int x,int y,double r, const string&
lbl)
{
    radius = r;
    label = lbl;
    cout << "Circle: " << label
        << " is created(int,int,double,string)\n";
}
void Circle::print(){
    cout << label << ".print -> at ";
    cout << "(" << getX() << ", " << getY() << ")\n";
    cout << " with radius = " << radius
        << endl;
}
Circle::~Circle(){
    cout << "Circle: " << label
        << " is destroyed"
        << endl;
}
```

```
int main()
{
    Shape s;
    s.print();
    Circle c;
    c.print();

    return 0;
}
```

```
Shape is created()
Shape.print ->(0,0)
Shape is created()
Circle: C is created()
C.print -> at (0,0) with radius = 1
Circle: C is destroyed
Shape is destroyed
Shape is destroyed
```

Passing arguments to the Base class Constructor

Allows selection between multiple base class constructors

Can also be done with inline constructors

Must be done if base class has no default constructor

```
class Parent {  
    string label;  
public:  
    Parent() {  
        label = "default";  
        cout << label  
            << "is created\n";  
    }  
    Parent(string val) {  
        label = val;  
        cout << label  
            << "is created\n";  
    }  
    string getLabel() {  
        return label;  
    }  
    ~Parent() {  
        cout << label  
            << " destroyed \n";  
    }  
};
```

```
class Child: public Parent{  
public:  
    Child(){  
        cout << "Child()"  
            << endl;  
    }  
    Child(string v):Parent(v)  
    {  
        cout << "Child(string)"  
            << endl;  
    }  
    ~Child(){  
        cout << getLabel()  
            << " destroyed"  
            << endl;  
    }  
};
```

Important

- In a derived class, some constructors can be inherited from the base class.
- The constructors that cannot be inherited are:
 - the default constructor
 - the copy constructor
 - the move constructor

```
int main()  
{  
    Child c("Superman");  
  
    return 0;  
}
```

```
Supermanis created  
Child(string)  
Superman destroyed  
Superman destroyed
```

Constructor Inheritance

```
class MyBase
{
private:
    int ival;
    double dval;
public:
    MyBase(int i)
    { ival = i; }

    MyBase(double d)
    { dval = d; }
};
```

```
class MyDerived : MyBase
{
public:
    MyDerived(int i) : MyBase(i)
    {}

    MyDerived(double d) : MyBase(d)
    {}
};
```



```
class MyDerived : MyBase
{
    using MyBase::MyBase;
};
```

The using statement causes the class to inherit the base class constructors.

Passing arguments to the Base class Constructor

```
class Parent {  
    string label;  
public:  
    Parent(){  
        label = "default";  
        cout << label  
            << "is created"  
            << endl;  
    }  
    Parent(string val){  
        label = val;  
        cout << label  
            << "is created"  
            << endl;  
    }  
    string getLabel(){  
        return label;  
    }  
    ~Parent(){  
        cout << label  
            << " destroyed "  
            << endl;  
    }  
};
```

```
class Child: public Parent{  
public:  
    using Parent::Parent;  
  
    ~Child(){  
        cout << getLabel()  
            << " destroyed"  
            << endl;  
    }  
};
```

```
int main()  
{  
    Child c("Superman");  
  
    return 0;  
}
```

```
Superman is created(string)  
Superman destroyed  
Superman destroyed
```

Redefining base class methods (Overriding)

Redefining methods: methods in a derived class that has the same name and parameter list as a method in the base class

Used to replace a method in base class with different actions in derived class

Not the same as overloading – with overloading, parameter lists must be different and exists in the same class

Objects of base class use base class version of the method

Objects of derived class use derived class version of method

```
class Parent{
    string label;
public:
    Parent():label("default"){ }
    Parent(string val):label(val){ }
    string getLabel(){return label;}
    void print() {
        cout << "Parent ...\n";
    }
};
```

```
class Child: public Parent{
public:
    Child():Parent(){ }
    Child(string v):Parent(v){ }
};
```

```
class Child: public Parent {
public:
    Child():Parent(){ }
    Child(string v):Parent(v){ }
    void print() {
        cout << "Child ..." << endl;
    }
};
```

Parent ...
Child ...

```
int main() {
    Parent p("Superman");
    Child c("Hulk");
    p.print();
    c.print();
    return 0;
}
```

Parent ...
Parent ...

Problems with redefining

Consider this situation:

Class Parent defines functions printX() and printY(). PrintX() calls printY().

Class Child inherits from Parent and redefines method printY().

An object child of class Child is created and method printX() is called.

When printX() is called, which y() is used, the one defined in Parent or the the redefined one in Child?

```
int main() {  
    Child child("Hulk");  
    child.printX();  
  
    return 0;  
}
```

```
Parent A...  
Parent B...
```

```
class Parent{  
    string label;  
public:  
    Parent():label("default"){  
    Parent(string val):label(val){  
    string getLabel(){return label;}  
    void printY(){  
        cout << "Parent A..." << endl;  
    }  
    void printX(){  
        printY();  
        cout << "Parent B..." << endl;  
    }  
};
```

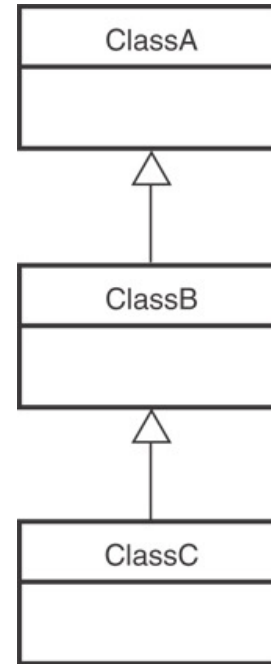
```
class Child: public Parent{  
public:  
    Child():Parent(){ }  
    Child(string v):Parent(v){ }  
    void printY(){  
        cout << "Child A..." << endl;  
    }  
};
```

Class Hierarchies

A base class can be derived from another base class.

```
int main()
{
    cout << "-----1-----" << endl;
    A a;
    cout << "-----2-----" << endl;
    B b;
    cout << "-----3-----" << endl;
    C c;
    cout << "-----4-----" << endl;
    return 0;
}
```

```
-----1-----
A Const
-----2-----
A Const
B Const
-----3-----
A Const
B Const
C Const
-----4-----
C Destr
B destr
A destr
B destr
A destr
A destr
```



```
class A{
public:
    A(){
        cout << "A Const" << endl;
    }
    ~A(){
        cout << "A destr" << endl;
    }
};

class B:public A{
public:
    B(){
        cout << "B Const" << endl;
    }
    ~B(){
        cout << "B destr" << endl;
    }
};

class C:public B{
public:
    C(){
        cout << "C Const" << endl;
    }
    ~C(){
        cout << "C Destr" << endl;
    }
};
```

Class Hierarchies and method redefining

```
int main()
{
    A a;
    B b;
    C c;
    a.print();
    b.print();
    c.print();

    return 0;
}
```

```
A..
B..
C..
```

```
class A{
public:
    void print(){
        cout << "A.." << endl;
    }
};

class B:public A{
public:
    void print(){
        cout << "B.." << endl;
    }
};

class C:public B{
public:
    void print(){
        cout << "C.." << endl;
    }
};
```

Polymorphism and Virtual Member Functions

Virtual member function: method in base class that expects to be redefined in derived class

method defined with key word virtual:

```
virtual void printY() {...}
```

Supports dynamic binding: functions bound at run time to function that they call

Without virtual member functions, C++ uses static (compile time) binding

```
class A{
public:
    void print(){
        cout << "A.." << endl;
    }
};
class B:public A{
public:
    void print(){
        cout << "B.." << endl;
    }
};
```

```
class C:public B{
public:
    void print(){
        cout << "C.." << endl;
    }
};
```

```
int main()
{
    A *aa = new A(); aa->print();
    A *ab = new B(); ab->print();
    A *ac = new C(); ac->print();
    B *bb = new B(); bb->print();
    B *bc = new C(); bc->print();
    C *cc = new C(); cc->print();
    return 0;
}
```

```
A..
A..
A..
B..
B..
C..
```

Polymorphism and Virtual Member Functions

```
class A{  
public:  
    virtual void print(){  
        cout << "A.." << endl;  
    }  
};
```

```
class B:public A{  
public:  
    void print(){  
        cout << "B.." << endl;  
    }  
};
```

```
class C:public B{  
public:  
    void print(){  
        cout << "C.." << endl;  
    }  
};
```

```
int main()  
{  
    A *aa = new A(); aa->print();  
    A *ab = new B(); ab->print();  
    A *ac = new C(); ac->print();  
    B *bb = new B(); bb->print();  
    B *bc = new C(); bc->print();  
    C *cc = new C(); cc->print();  
    return 0;  
}
```

```
A..  
B..  
C..  
B..  
C..  
C..
```

Polymorphism and Virtual Member Functions

```
class A{
public:
    virtual void print(){
        cout << "A.." << endl;
    }
};

class B:public A{
public:
    virtual void print(){
        cout << "B.." << endl;
    }
};

class C:public B{
public:
    void print(){
        cout << "C.." << endl;
    }
};
```

```
int main()
{
    A *aa = new A(); aa->print();
    A *ab = new B(); ab->print();
    A *ac = new C(); ac->print();
    B *bb = new B(); bb->print();
    B *bc = new C(); bc->print();
    C *cc = new C(); cc->print();
    return 0;
}
```

```
A..
B..
C..
B..
C..
C..
```


Polymorphism and Virtual Member Functions

```
class A{
public:
    void print(){
        cout << "A.." << endl;
    }
};

class B:public A{
public:
    virtual void print(){
        cout << "B.." << endl;
    }
};

class C:public B{
public:
    void print(){
        cout << "C.." << endl;
    }
};
```

```
int main()
{
    A *aa = new A(); aa->print();
    A *ab = new B(); ab->print();
    A *ac = new C(); ac->print();
    B *bb = new B(); bb->print();
    B *bc = new C(); bc->print();
    C *cc = new C(); cc->print();
    return 0;
}
```

```
A..
A..
A..
B..
C..
C..
```

Virtual Member Functions

A virtual method is dynamically bound to calls at run-time.

At run-time, C++ determines the type of object making the call, and binds the method to the appropriate version of the method.

To make a method virtual, place the virtual key word before the return type in the base class's declaration:

```
virtual return_type method_name();
```

The compiler will not bind the method to calls. Instead, the program will bind them at run-time.

Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer, as demonstrated in the print() method.

Base class pointers

Can define a pointer to a base class object

Can assign it the address of a derived class object

B..
C..
C..
B..
C..

```
int main()
{
    A *ab = new B();
    ab->print();
    A *ac = new C();
    ac->print();
    B *bc = new C();
    bc->print();

    A *ptrb,*ptrc;
    B b;
    C c;
    ptrb = &b;
    ptrb->print();

    ptrc = &c;
    ptrc->print();

    return 0;
}
```

```
class A{
public:
    virtual void print() const{
        cout << "A.." << endl;
    }
};

class B:public A{
public:
    virtual void print() const{
        cout << "B.." << endl;
    }
};

class C:public B{
public:
    void print() const {
        cout << "C.." << endl;
    }
};
```

Base class pointers

Base class pointers and references only know about members of the base class
So, you can't use a base class pointer to call a derived class method

Redefined methods in derived class will be ignored unless base class declares the method virtual

In C++, redefined functions are statically bound and overridden functions are dynamically bound.

So, a virtual function is overridden, and a non-virtual function is redefined.

Virtual Destructors

It's a good idea to make destructors virtual if the class could ever become a base class.

Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from.

```
int main()
{
    cout << "...1..." << endl;
    A *a = new A();
    cout << "...2..." << endl;
    delete a;
    cout << "...3..." << endl;
    A *b = new B();
    cout << "...4..." << endl;
    delete b;
    cout << "...5..." << endl;

    return 0;
}
```

```
...1...
A constructed..
...2...
A Destroyed..
...3...
A constructed..
B constructed..
...4...
A Destroyed..
...5...
```

Child object is
not destroyed

```
class A{
public:
    A() {
        cout << "A constructed.." << endl;
    }
    ~A(){
        cout << "A Destroyed.." << endl;
    }
};
class B:public A{
public:
    B(){
        cout << "B constructed.." << endl;
    }
    ~B(){
        cout << "B Destroyed.." << endl;
    }
};
```

Virtual Destructors

```
int main()
{
    cout << "...1..." << endl;
    A *a = new A();
    cout << "...2..." << endl;
    delete a;
    cout << "...3..." << endl;
    A *b = new B();
    cout << "...4..." << endl;
    delete b;
    cout << "...5..." << endl;

    return 0;
}
```

Child object is
destroyed

```
...1...
A constructed..
...2...
A Destroyed..
...3...
A constructed..
B constructed..
...4...
B Destroyed..
A Destroyed..
...5...
```

```
class A{
public:
    A() {
        cout << "A constructed.." << endl;
    }
    virtual ~A(){
        cout << "A Destroyed.." << endl;
    }
};

class B:public A{
public:
    B(){
        cout << "B constructed.." << endl;
    }
    ~B(){
        cout << "B Destroyed.." << endl;
    }
};
```

Final methods

When a member function is declared with the final key word, it cannot be overridden in a derived class.

```
class A{
public:
    void print() {
        cout << "I am A" << endl;
    }
};

class B:public A {
public:
    void print(){
        cout << "I am B" << endl;
    }
};
```

I am A

```
int main()
{
    A *a = new B();
    a->print();
    return 0;
}
```

```
class A{
public:
    void print() final {
        cout << "I am A" << endl;
    }
};

class B:public A {
public:
    void print(){
        cout << "I am B" << endl;
    }
};
```

What is the problem?

```
g++ -c -g -MMD -MP -MF "build/Debug/GNU-
Linux/main.o.d" -o build/Debug/GNU-Linux/main.o
main.cpp
main.cpp:7:10: error: 'void A::print()' marked 'final',
but is not virtual
void print()final {
```

Final class

When a class is declared with the final key word, it cannot be inherited.

```
class A{
public:
    void print() {
        cout << "I am A" << endl;
    }
};

class B:public A {
public:
    void print(){
        cout << "I am B" << endl;
    }
};
```

```
int main()
{
    A *a = new B();
    a->print();
    return 0;
}
```

```
class A final {
public:
    void print() {
        cout << "I am A" << endl;
    }
};

class B:public A {
public:
    void print(){
        cout << "I am B" << endl;
    }
};
```

```
g++ -c -g -MMD -MP -MF "build/Debug/GNU-
Linux/main.o.d" -o build/Debug/GNU-Linux/main.o
main.cpp
main.cpp:11:7: error: cannot derive from 'final' base
'A' in derived type 'B'
class B:public A {
```


Abstract base classes and pure virtual functions

Pure virtual function: a virtual member function that must be overridden in a derived class that has objects

Abstract base class contains at least one pure virtual function:

```
virtual void Y() = 0;
```

The = 0 indicates a pure virtual function

Must have no function definition in the base class

```
int main()
{
    A *aa = new A();

    return 0;
}
```

Generates a compilation error. It is not possible to instantiate class A.

```
int main()
{
    A *ab = new B();
    ab->print();
    A *ac = new C();
    ac->print();

    return 0;
}
```

```
I am B
I am C
```

```
class A {
public:
    virtual void print() = 0;
};

class B:public A {
public:
    void print(){
        cout << "I am B" << endl;
    }
};

class C:public A {
public:
    void print(){
        cout << "I am C" << endl;
    }
};
```

Any class inherits from class A must implement the print method. Otherwise, a compilation error is generated.

Abstract base classes and pure virtual functions

Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects.

A class becomes an abstract base class when one or more of its member functions is a pure virtual function.

```
class Shape {  
private:  
    int x;  
    int y;  
    string label;  
public:  
    Shape();  
    Shape(int,int);  
    Shape(int,int, string);  
    virtual void print() = 0;  
    void setX(int);  
    void setY(int);  
    int getX();  
    int getY();  
    ~Shape();  
};
```

Shape is an abstract class
Print is a virtual function.

```
class Circle:public Shape {  
private:  
    double radius;  
    string label;  
public:  
    Circle();  
    Circle(int,int);  
    Circle(int, int, double, const string&);  
    void print();  
    void setCenter(int,int);  
    void setRadius(double r);  
    double getRadius();  
    ~Circle();  
};
```

Circle must implement the print function. Otherwise, a compilation error is generated.

```
void Circle::print(){  
    cout << label << ".print -> at ";  
    cout << "(" << getX() << ", "  
        << getY() << ")";  
    cout << " with radius = "  
        << radius  
        << endl;  
}
```

```
int main() {  
    Circle c;  
    c.print();  
    return 0;  
}
```

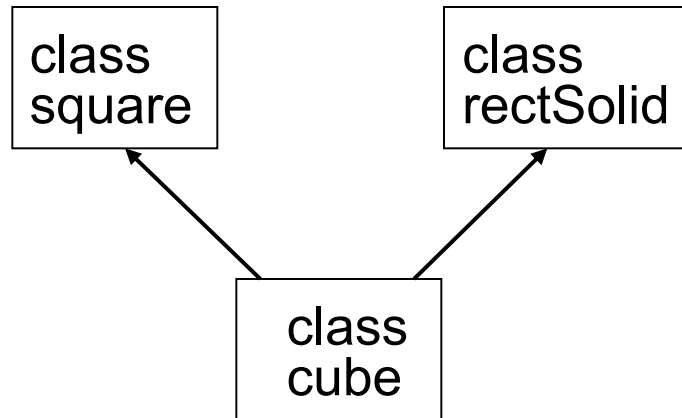
Shape is created()
Circle: C is created()
C.print -> at (0,0) with radius = 1
Circle: C is destroyed
Shape is destroyed

Multiple Inheritance

A derived class can have more than one base class

Each base class can have its own access specification in derived class's definition:

```
class cube : public square, public rectSolid;
```



Multiple Inheritance – base class constructors execution order

```
class A{
public:
A(){cout << "A const def\n";}
~A(){cout << "A destr\n";}
};
class B{
public:
B(){cout << "B const def\n";}
~B(){cout << "B destr\n";}
};

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

```
class A{
public:
A(){cout << "A const def\n";}
~A(){cout << "A destr\n";}
};
class B{
public:
B(){cout << "B const def\n";}
~B(){cout << "B destr\n";}
};

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

A const def
B const def
B destr
A destr

```
int main()
{
    C c;
    return 0;
}
```

B const def
A const def
A destr
B destr

Multiple Inheritance – base class constructors execution order (2)

```
class A{
public:
    A(){cout << "A const def\n";}
    ~A(){cout << "A destr\n";}
};

class B{
public:
    B(){cout << "B const def\n";}
    ~B(){cout << "B destr\n";}
};


class C:public B, public A{
public:
    C(){cout << "C const def\n";}
    ~C(){cout << "C destr\n";}
};
```

```
int main()
{
    C c;
    return 0;
}
```

```
int main() {
    C *p = new C();
    p->print();

    delete p;
    return 0;
}
```

```
B const def
A const def
C const def
C destr
A destr
B destr
```



Super class constructors first
Constructors of B before A

Multiple Inheritance – base class methods

```
class A{
public:
    A(){cout << "A const def\n";}
    void printA(){
        cout << "I am A\n";
    }
    ~A(){cout << "A destr\n";}
};
class B{
public:
    B(){cout << "B const def\n";}
    void printB(){
        cout << "I am B\n";
    }
    ~B(){cout << "B destr\n";}
};
```

```
class C:public B, public A{
public:
    C(){cout << "C const def\n";}
    void printC(){
        printA();
        printB();
        cout << "I am C\n";
    }
    ~C(){cout << "C destr\n";}
};
```

PrintA() and printB() both are inherited in C.

- ✘ Public and protected members of A and B are inherited by C

```
int main()
{
    C c;
    c.printC();

    return 0;
}
```

```
B const def
A const def
C const def
I am A
I am B
I am C
C destr
A destr
B destr
```

Multiple Inheritance – base class methods (2)

```
class A{
public:
    A(){cout << "A const def\n";}
    void print () {
        cout << "I am A\n";
    }
    ~A(){cout << "A destr\n";}
};

class B{
public:
    B(){cout << "B const def\n";}
    void print () {
        cout << "I am B\n";
    }
    ~B(){cout << "B destr\n";}
};
```

```
class C:public B, public A{
public:
    C(){cout << "C const def\n";}
    void print () {
        A::print();
        B::print();
        cout << "I am C\n";
    }
    ~C() {cout << "C
destr\n";}
};
```

When methods are redefined in the subclass, the super class name can be used with the scope operator :: to identify which method is called.

```
int main()
{
    C c;
    c.print ();

    return 0;
}
```

```
B const def
A const def
C const def
I am A
I am B
I am C
C destr
A destr
B destr
```

Multiple Inheritance – base class methods (2)

```
class A{
public:
    A(){cout << "A const def\n";}
    void print () {
        cout << "I am A\n";
    }
    ~A(){cout << "A destr\n";}
};

class B{
public:
    B(){cout << "B const def\n";}
    void print () {
        cout << "I am B\n";
    }
    ~B(){cout << "B destr\n";}
};
```

```
class C:public B, public A{
public:
    C(){cout << "C const def\n";}
    void print () {
        A::print();
        B::print();
        cout << "I am C\n";
    }
    ~C() {cout << "C
destr\n";}
};
```

```
int main() {
    A *pa = new C();
    B *pb = new C();
    pa->print();
    pb->print();
    delete pa;
    delete pb;
    return 0;
}
```

Delete pa
invoked the
destructor of
the parent class
A not the one in
class C.

✗ Delete pb
invoked the
destructor of
the parent class
B not the one in
class C.

```
B const def
A const def
C const def
B const def
A const def
C const def
I am A
I am B
A destr
B destr
```


Multiple Inheritance – virtual destructors

```
class A{
public:
    A(){cout << "A const def\n";}
    void print () {
        cout << "I am A\n";
    }
    virtual ~A(){cout << "A destr\n";}
};

class B{
public:
    B(){cout << "B const def\n";}
    void print () {
        cout << "I am B\n";
    }
    virtual ~B(){cout << "B destr\n";}
};
```

```
class C:public B, public A{
public:
    C(){cout << "C const def\n";}
    void print () {
        cout << "I am C\n";
    }
    ~C() {cout << "C destr\n";}
};
```

```
int main() {
    A *pa = new C();
    B *pb = new C();
    pa->print();
    pb->print();
    delete pa;
    delete pb;
    return 0;
}
```

```
B const def
A const def
C const def
B const def
A const def
C const def
I am A
I am B
C destr
A destr
B destr
C destr
A destr
B destr
```

- When adding the virtual keyword to the destructors of the super class destructor, this will invoke a full cleanup that will cause the destructors of the super class to be invoked.

Multiple Inheritance – virtual methods

```
class A{
public:
    A(){cout << "A const def\n";}
    virtual void print () {
        cout << "I am A\n";
    }
    virtual ~A(){cout << "A destr\n";}
};

class B{
public:
    B(){cout << "B const def\n";}
    virtual void print () {
        cout << "I am B\n";
    }
    virtual ~B(){cout << "B destr\n";}
};
```

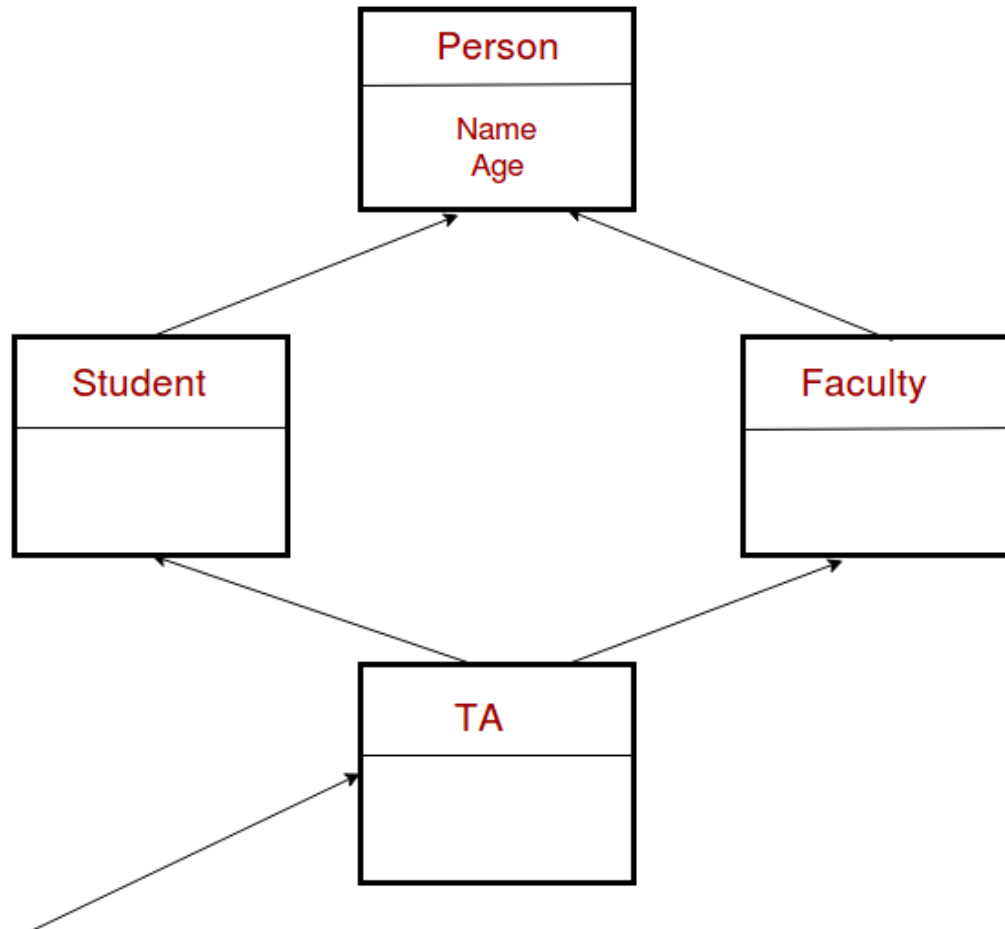
```
class C:public B, public A{
public:
    C(){cout << "C const def\n";}
    void print () {
        cout << "I am C\n";
    }
    ~C() {cout << "C destr\n";}
};
```

```
int main() {
    A *pa = new C();
    B *pb = new C();
    pa->print();
    pb->print();
    delete pa;
    delete pb;
    return 0;
}
```

```
B const def
A const def
C const def
B const def
A const def
C const def
I am C
I am C
C destr
A destr
B destr
C destr
A destr
B destr
```

■ Virtual keyword before the print method will cause the compiler to postpone the binding of which print to call until run time. Causing it to call the print method inside the C class.

The Diamond Problem



Name and Age needed only once

The Diamond Problem

```
class Person {
private:
    string name;
    int age;
public:
    Person():name("X"),age(0) {
cout << name << " Person::def\n" ;
    }
    Person(string nm, int g):name(nm),age(g) {
cout << name << " Person::none def\n";
    }
    string getName() { return name; }
    ~Person(){ cout << name << " ~Person\n"; }
};
```

```
class TA:public Faculty,public Student {
public:
    TA():Student(), Faculty() {
        cout<< Student::getName()
        <<" TA::none default"<< endl;
    }
    TA(string nm, int g):Student(nm,g), Faculty(nm,g) {
        cout<< Faculty::getName()
        <<" TA::none default"<< endl;
    }
    ~TA(){ cout << Student::getName() << " ~TA\n"; }
};
```

```
class Student:public Person {
public:
    Student():Person() {
        cout << getName() << " Student::default\n";
    }
    Student(string nm, int g):Person(nm, g) {
        cout << nm << " Student::none default \n";
    }
    ~Student(){
        cout << getName() << " ~Student\n";
    }
};
```

```
class Faculty:public Person {
public:
    Faculty():Person() {
        cout << getName() << " Faculty::default\n";
    }
    Faculty(string nm, int g):Person(nm, g) {
        cout << nm << " Faculty::none default \n";
    }
    ~Faculty(){
        cout << getName() << " ~Faculty\n";
    }
};
```

The Diamond Problem

```
int main(){  
    TA ta1("Sharaf",50);  
  
    return 0;  
}
```

```
Sharaf Person::none def  
Sharaf Faculty::none default  
Sharaf Person::none def  
Sharaf Student::none default  
Sharaf TA::none default  
Sharaf ~TA  
Sharaf ~Student  
Sharaf ~Person  
Sharaf ~Faculty  
Sharaf ~Person
```

The constructor of 'Person' is called two times.

The Destructor of 'Person' is also called two times when object 'ta1' is destructed.

So object 'ta1' has two copies of all members of 'Person', this causes ambiguities.

The Diamond Problem: solution

By adding the 'virtual' keyword to the classes Faculty and Student, the issue of double creating a Person object when constructing the ta1 object will be resolved.

```
int main(){
    TA ta1("Sharaf",50);

    return 0;
}
```

```
X Person::def
Sharaf Faculty::none default
Sharaf Student::none default
X TA::none default
X ~TA
X ~Student
X ~Faculty
X ~Person
```

```
class Student: virtual public Person {
public:
    Student():Person() {
        cout << getName() << " Student::default\n";
    }
    Student(string nm, int g):Person(nm, g) {
        cout << nm << " Student::none default \n";
    }
    ~Student(){
        cout << getName() << " ~Student\n";
    }
};
```

```
class Faculty: virtual public Person {
public:
    Faculty():Person() {
        cout << getName() << " Faculty::default\n";
    }
    Faculty(string nm, int g):Person(nm, g) {
        cout << nm << " Faculty::none default \n";
    }
    ~Faculty(){
        cout << getName() << " ~Faculty\n";
    }
};
```

Override

The override key word tells the compiler that the function is supposed to override a function in the base class.

```
class A{
public:
    virtual void print() const {
        cout << "I am A" << endl;
    }
};

class B:public A {
public:
    void print() {
        cout << "I am B" << endl;
    }
};
```

I am A

What is the problem?

```
int main()
{
    A *a = new B();
    a->print();
    return 0;
}
```

```
class A{
public:
    virtual void print() const {
        cout << "I am A" << endl;
    }
};

class B:public A {
public:
    void print() override {
        cout << "I am B" << endl;
    }
};
```

```
g++ -c -g -MMD -MP -MF "build/Debug/GNU-
Linux/main.o.d" -o build/Debug/GNU-Linux/main.o
main.cpp
main.cpp:13:10: error: 'void B::print()' marked
'override', but does not override
    void print() override {
```