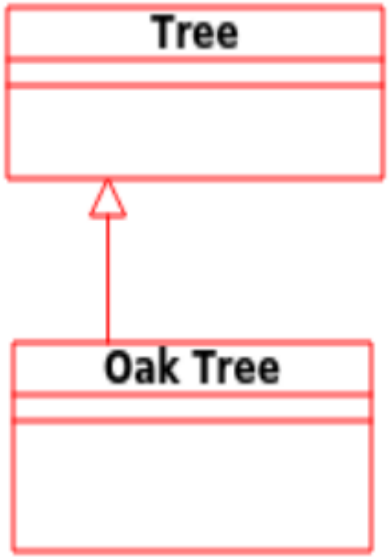
CSE 1325

Week of 11/16/2020

Instructor : Donna French

Object Relationships

Inheritance



- Represents the "is a" relationship
- Shows the relationship between a super class/base class/parent and a derived/subclass/child.
- Arrow is on the side of the base class/parent

Inheritance

Inheritance involves creating new objects by directly acquiring the attributes and behaviors of other objects and then extending or specializing them.

Inheritance is everywhere in real life.

Most living things (including you) inherited traits from parents.

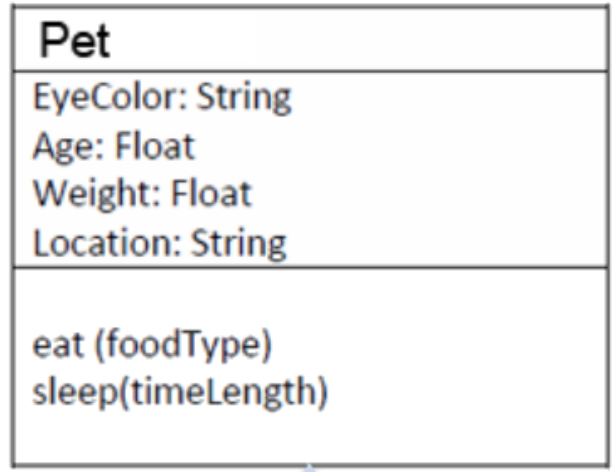
Even non living things can inherit traits from their predecessors.

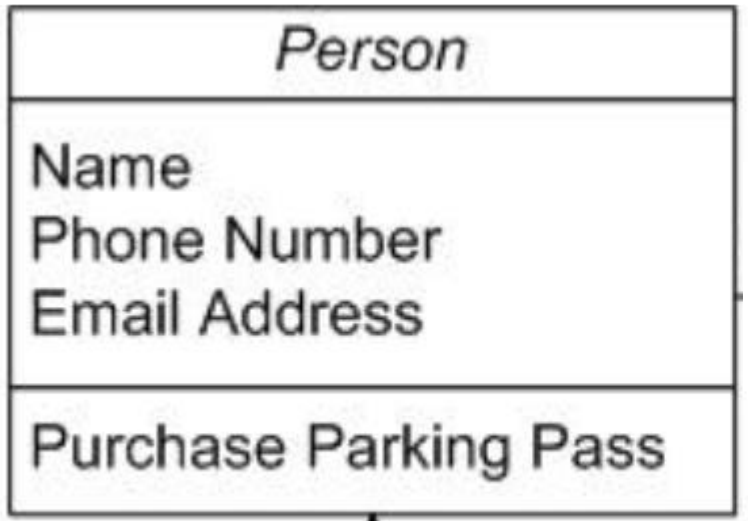
Inheritance

When Apple decides to create the next generation of iPhone, it does not start from scratch when creating a new phone.

They start with what they already know about the current version of the iPhone and build upon that.

Most new version of electronics build upon the previous version. Not only does this lead to less work to create a new version but it also allows for backward compatibility.


Pet
EyeColor: String
Age: Float
Weight: Float
Location: String
eat (foodType)
sleep(timeLength)


Person
Name
Phone Number
Email Address
Purchase Parking Pass

has a

is a

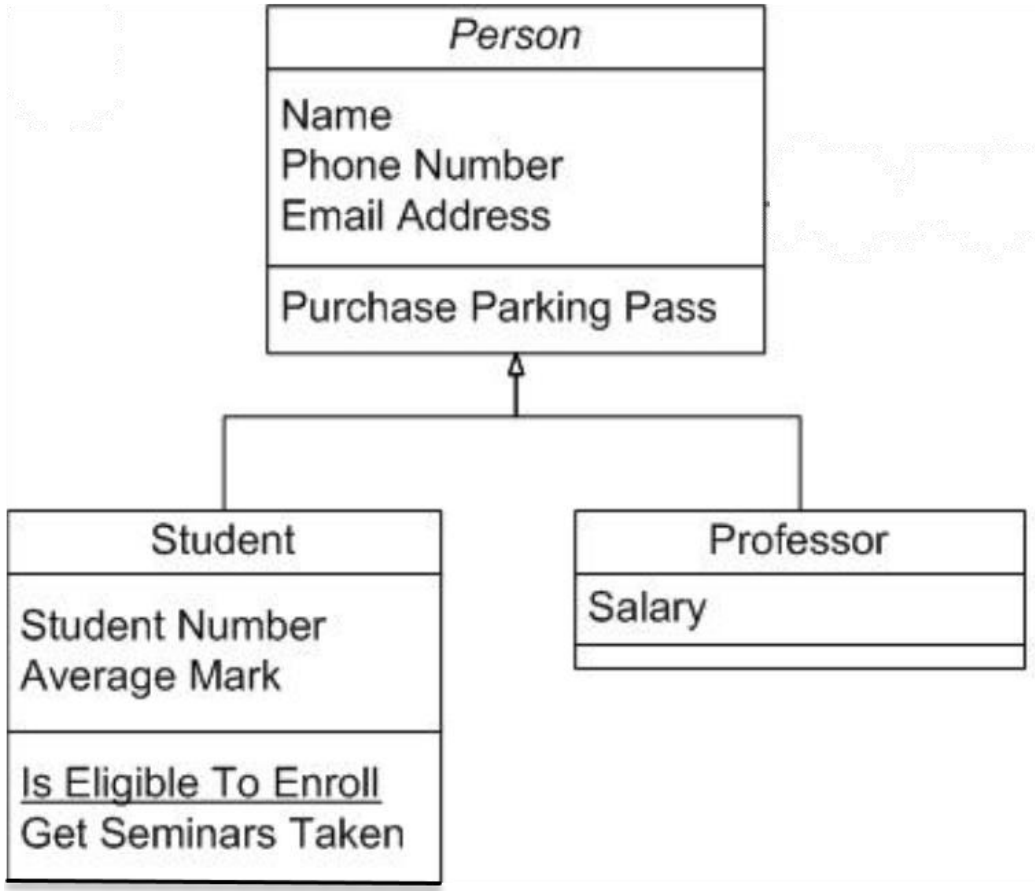
Inheritance

So does a student have a name, phone number and email address? Can a student purchase a parking pass?

Does a professor have a name, phone number and email address? Can a professor purchase a parking pass?

A professor has a salary – does every person have a salary?

A student has a student number – does every person have a student number?

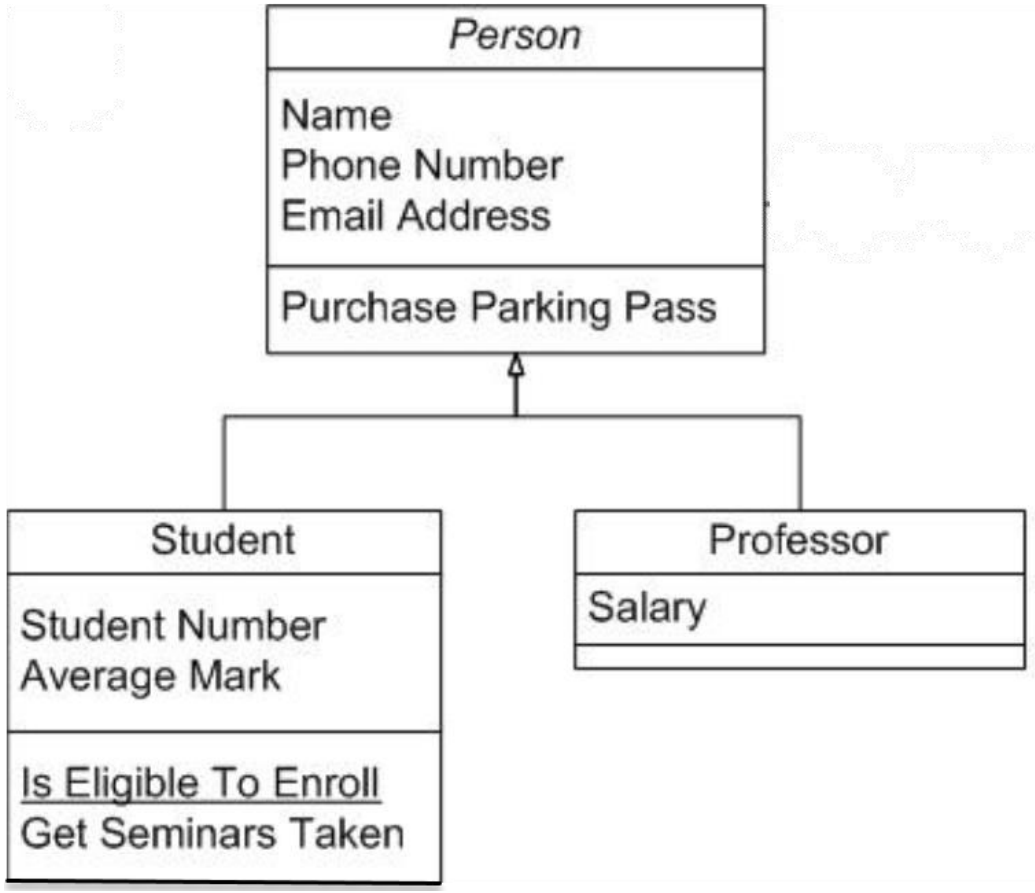


Inheritance

Rather than include name, phone number, email address and the ability to purchase a parking pass in our Student and Professor classes, we allow Student and Professor to inherit those attributes/abilities from Person.

This reduces the complexity of the Student and Professor class by making them contain less.

This also allows us to make changes to the Person class without directly changing Student and Professor.



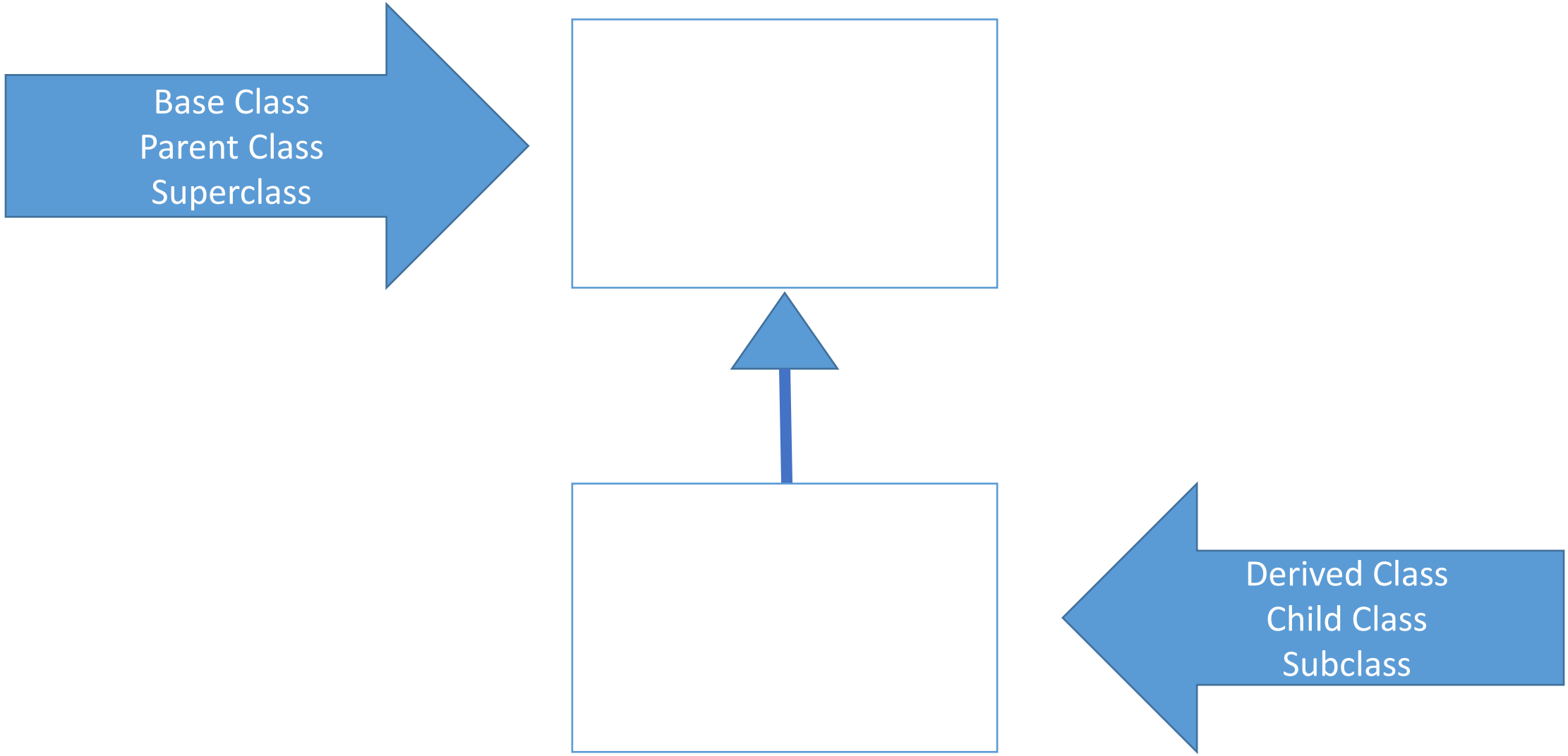
Inheritance

A form of software reuse in which you create a class that absorbs an existing class's data and behaviors and enhances them with new capabilities.

- You can designate that a new class should inherit the members of an existing class.

- This existing class is called the base class, and the new class is referred to as the derived class.

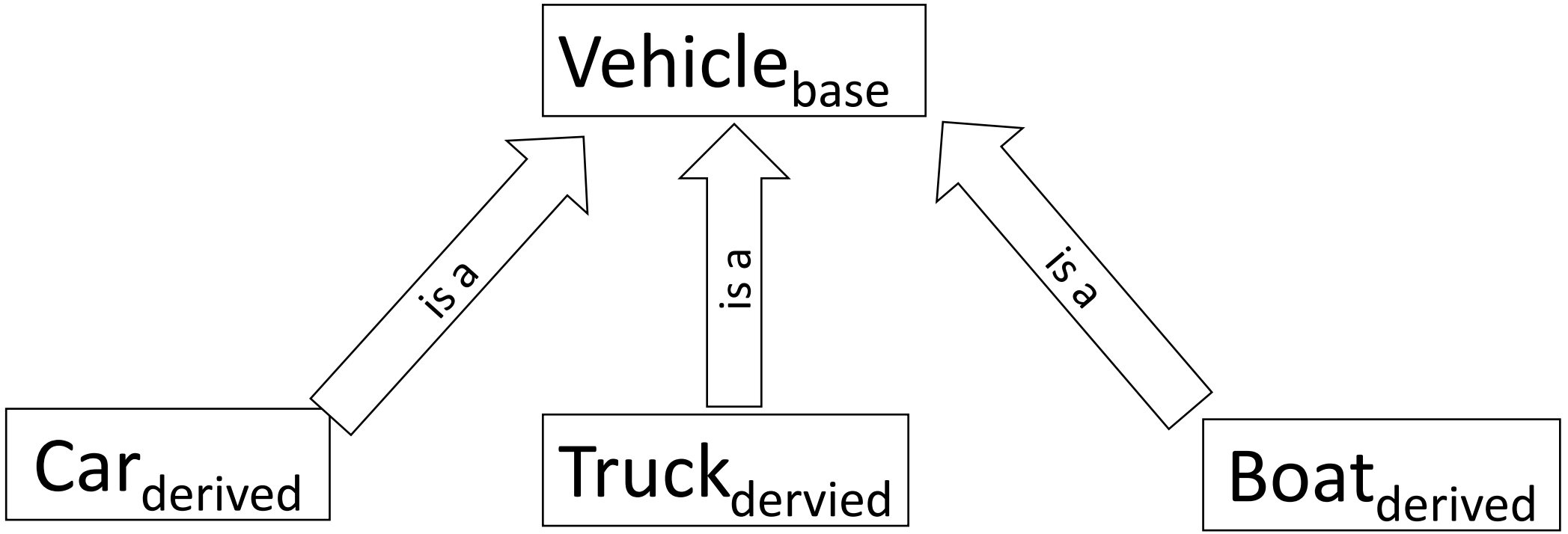
Inheritance



Inheritance

- A derived class represents a *more specialized* group of objects.
- C++ offers `public`, `protected` and `private` inheritance.
- With `public` inheritance, every object of a derived class is also an object of that derived class's base class.
- However, base-class objects are not objects of their derived classes.

base-class objects are not objects of their derived classes



every object of a derived class is also an object of that derived class's base class.

Inheritance

Base classes tend to be *more general* and derived classes tend to be *more specific*.

Base Class	**Derived Class**
Student	GraduateStudent, UnderGraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, StudentLoan
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

Inheritance

Because every derived-class object *is an* object of its base class and one base class can have *many* derived classes, the set of objects represented by a base class typically is *larger* than the set of objects represented by any of its derived classes.

Base class `Vehicle` represents all vehicles including cars, boats, trucks, airplanes and bicycles.

Derived class `Car` represents a smaller, more specific subset of all vehicles.

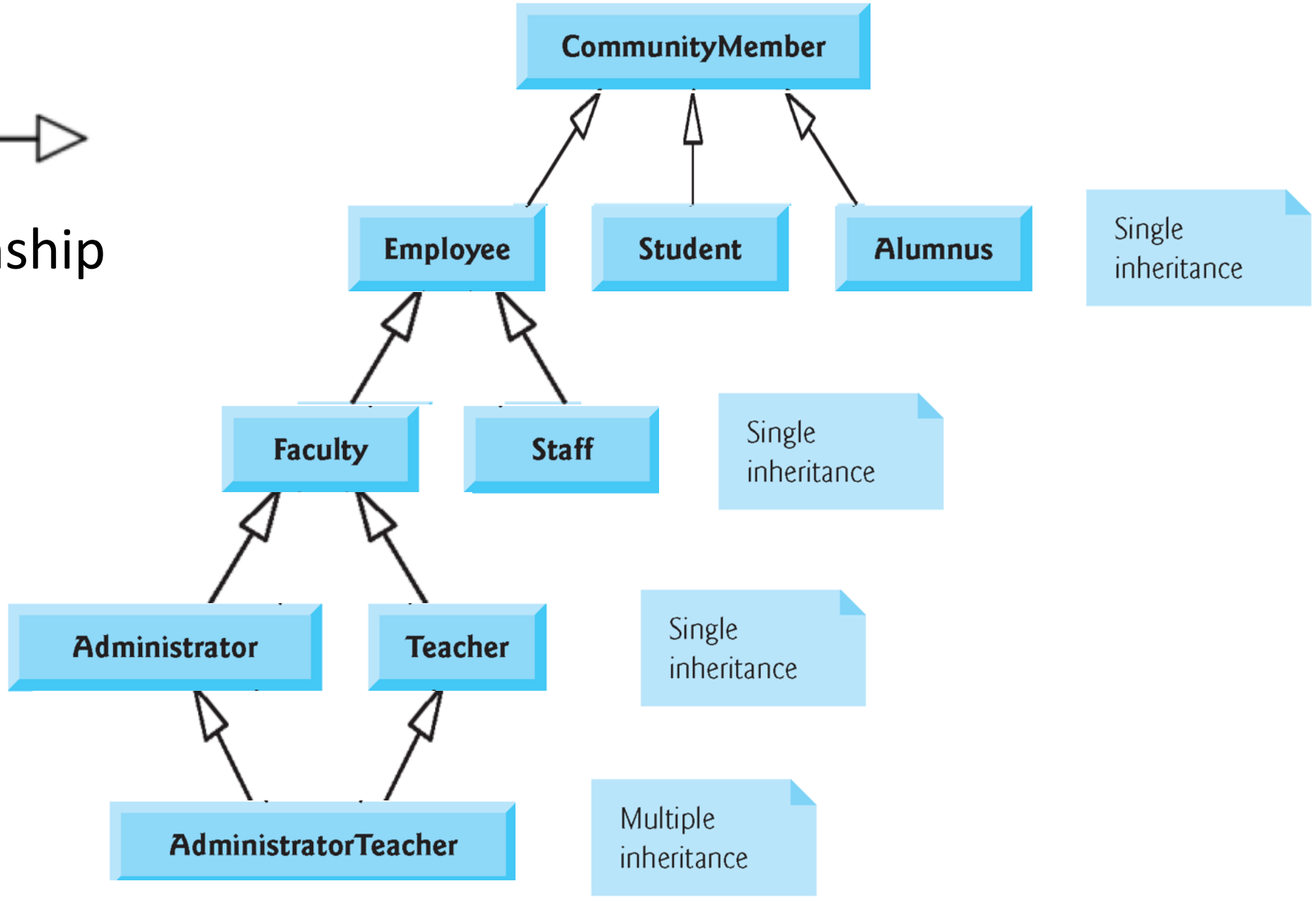
Base class `Pet` represents all animals kept as pets whereas as derived class `Cat` is a specific subset of `Pet`.

Inheritance

Inheritance relationships form class hierarchies.

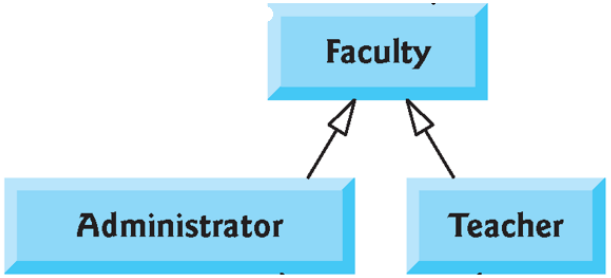
- A base class exists in a hierarchical relationship with its derived classes.
- Although classes can exist independently, once they are associated with an inheritance relationships, they become related to other classes.
- A class becomes either a base class—supplying members to other classes, a derived class—inheriting its members from other classes, or *both*.

is-a relationship



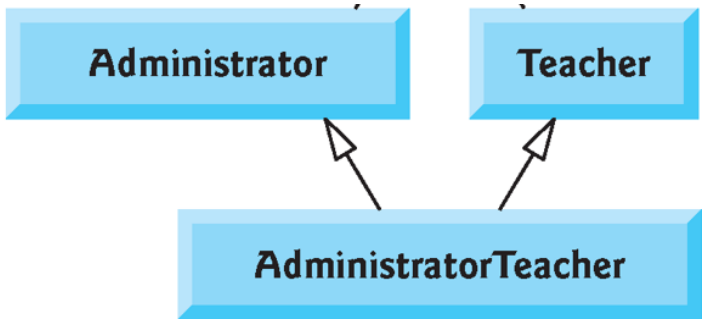
Inheritance

Single Inheritance



A class is derived from one base class

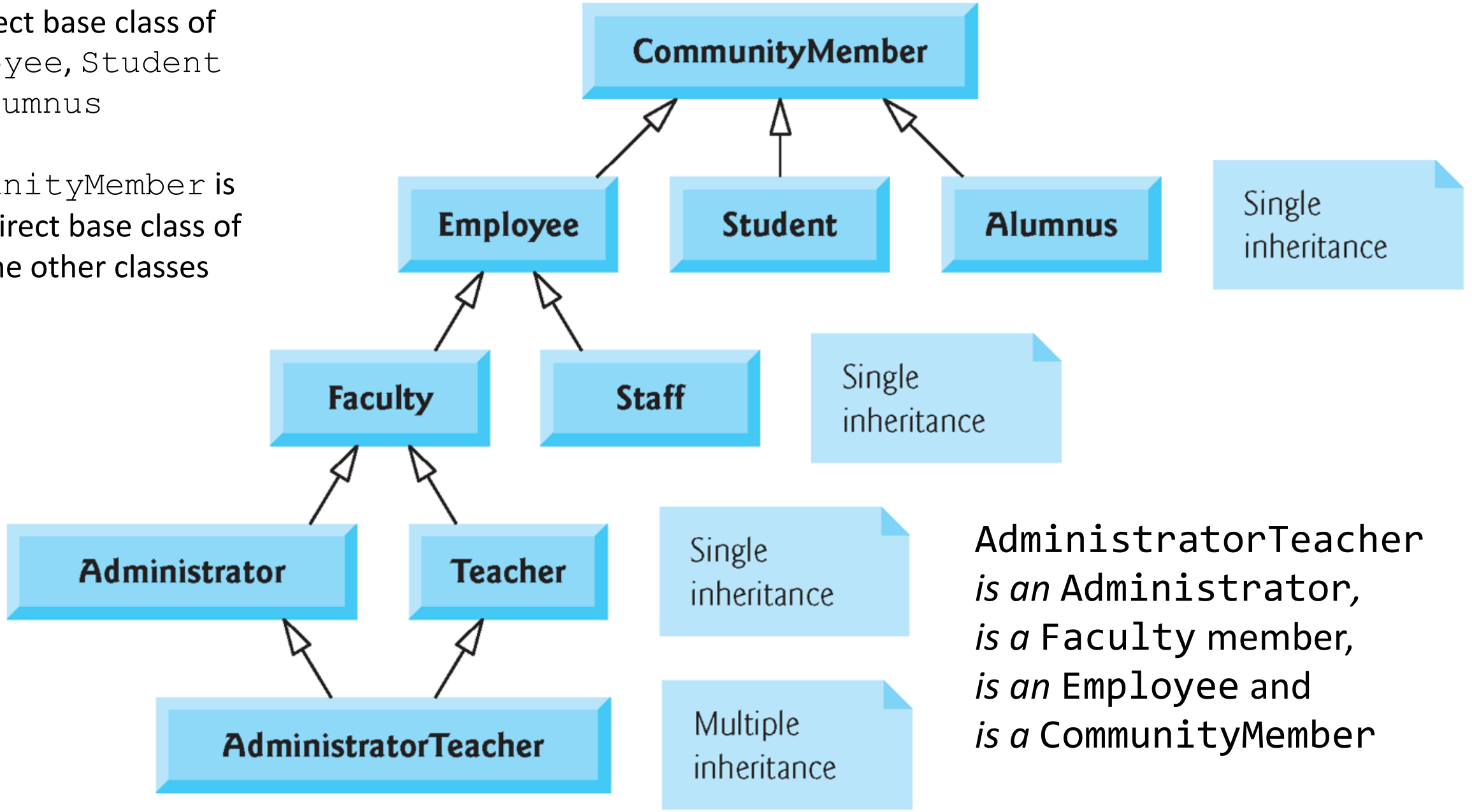
Multiple inheritance



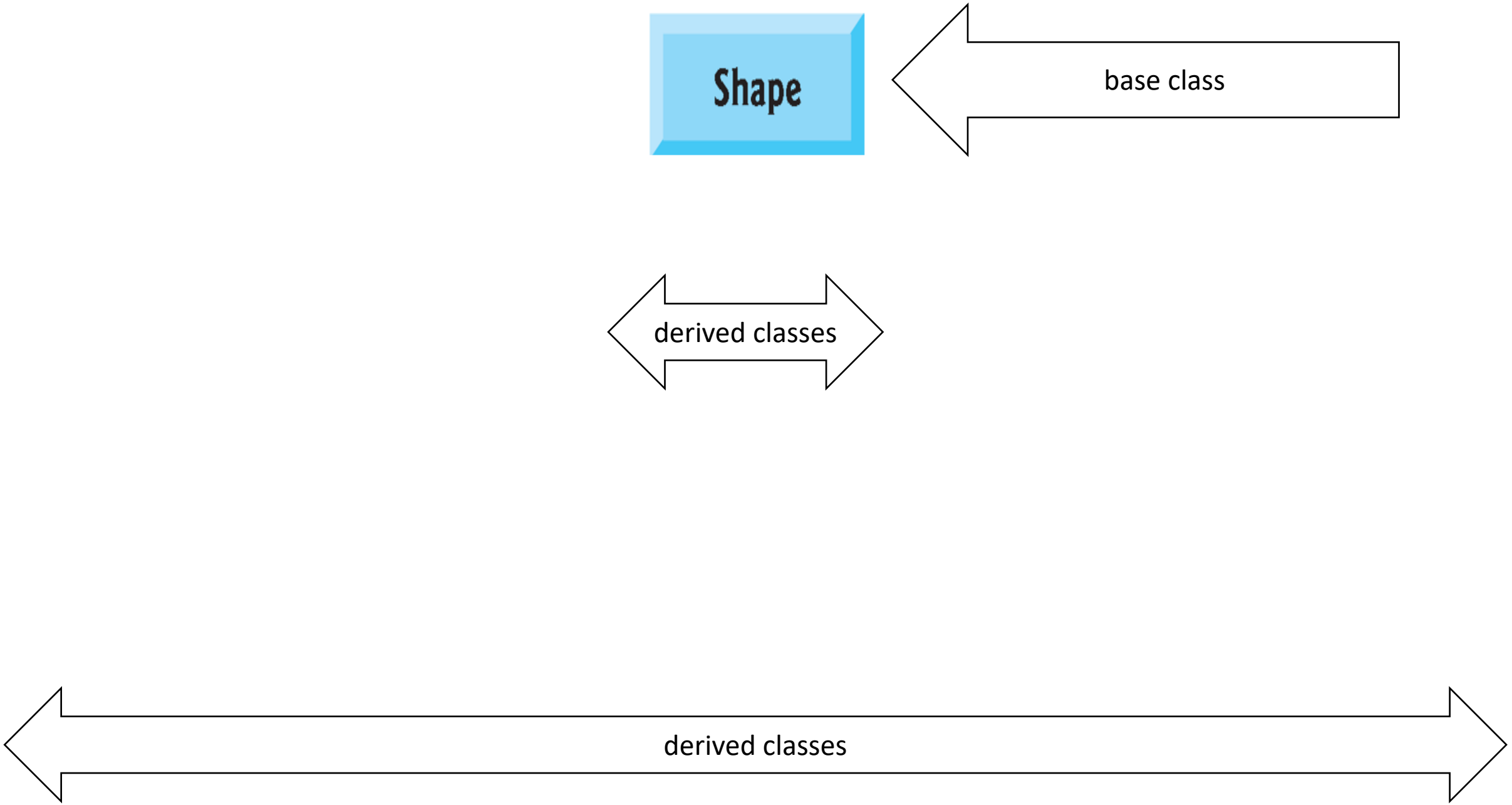
A derived class inherits simultaneously from two or more (possibly unrelated) base classes.

`CommunityMember` is the direct base class of `Employee`, `Student` and `Alumnus`

`CommunityMember` is the indirect base class of all of the other classes



`AdministratorTeacher` *is an* `Administrator`, *is a* `Faculty` member, *is an* `Employee` and *is a* `CommunityMember`


Shape
base class
derived classes
derived classes

Inheritance

3 forms of inheritance

- public
- private
- protected

Regardless of the form of inheritance, private data members of the base class are not accessible directly from that class's derived classes.

Private base-class data members are still inherited – still considered parts of the derived class

Inheritance

Public inheritance

all other base-class members retain their original member access then they become members of the derived class

public members of the base class become public members of the derived class

Inheritance

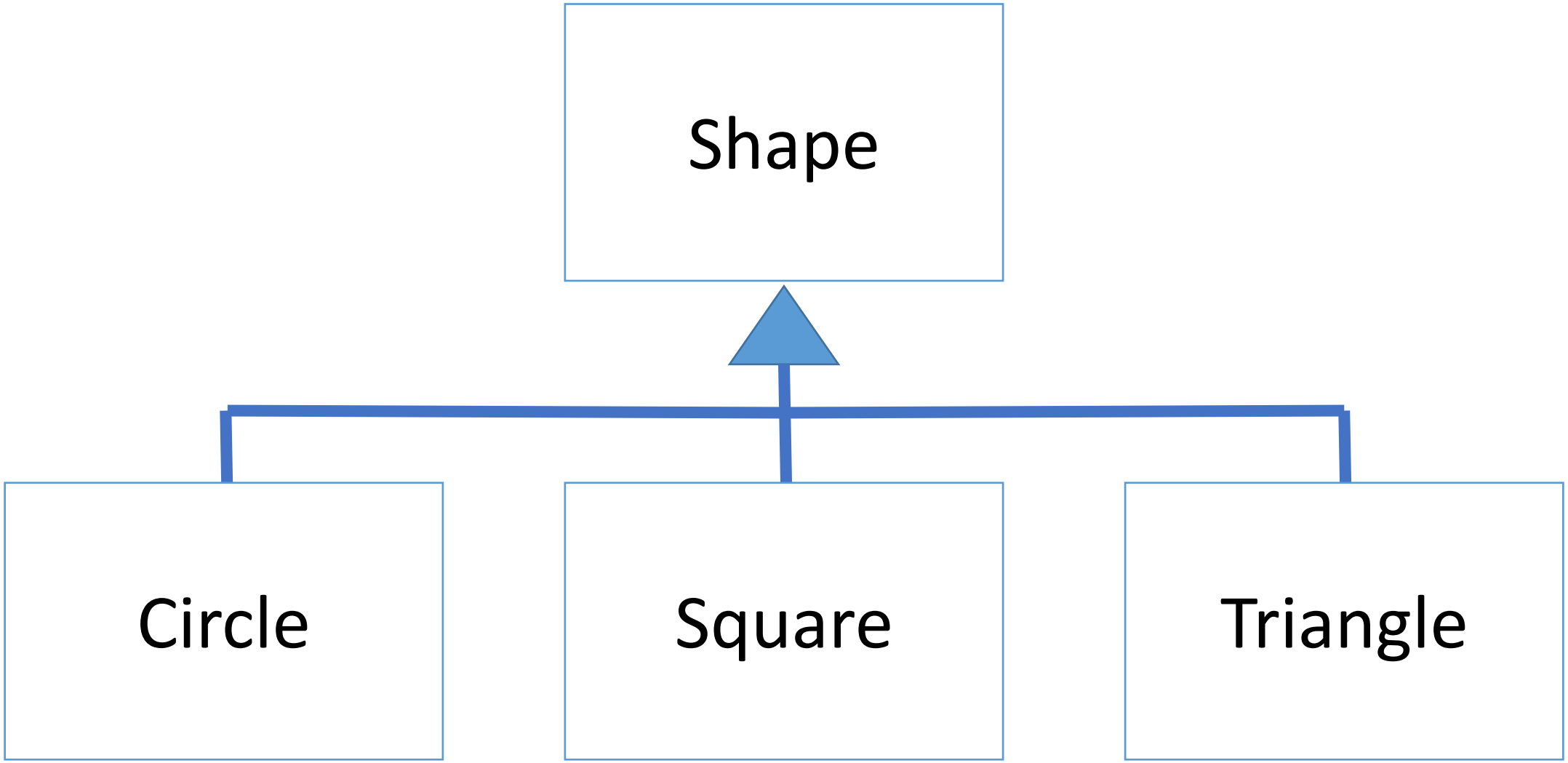
With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in the base class.

When changes are required for these common features, you need to make the changes only the base class.

Derived classes then inherit the changes.

Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

Inheritance



```
class Shape
{
  public :
    Shape(std::string name) : ShapeName{name}
    {
    }
    std::string getName()
    {
      return ShapeName;
    }

    float dim1;
    float dim2;
    std::string ShapeName;
};
```

```
int main (void)
{
   Shape A("Poly");

   std::cout << "My name is "
             << A.getName()
             << std::endl;

   return 0;
}
```

```
My name is Poly
```

Now I want to create a class Circle.

The more abstract version of Circle is Shape.

Shape knows its name and how to get its name. Shape also knows dimensions.

We want Circle to know these same things but we also want Circle to calculate its area.

When we create a Circle object, we want to construct it with its dimension/radius set already.

```
class Circle
{
  public:
    Circle(float radius=0)
    {
      dim1 = dim2 = radius;
    }

    float getarea()
    {
      return dim1 * dim2 * M_PI;
    }
};
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11  ShapeInherit.cpp -o ShapeInherit.o
ShapeInherit.cpp: In constructor 'Circle::Circle(float)':
ShapeInherit.cpp:28:3: error: no matching function for call to
'Shape::Shape()'
   {
   ^
ShapeInherit.cpp:11:3: note: candidate: Shape::Shape(std::__cxx11::string)
   Shape(std::string name) : ShapeName{name}
   ^~~~~
ShapeInherit.cpp:11:3: note:   candidate expects 1 argument, 0 provided
ShapeInherit.cpp:8:7: note: candidate: Shape::Shape(const Shape&)
 class Shape
       ^~~~~
ShapeInherit.cpp:8:7: note:   candidate expects 1 argument, 0 provided
ShapeInherit.cpp:8:7: note: candidate: Shape::Shape(Shape&&)
ShapeInherit.cpp:8:7: note:   candidate expects 1 argument, 0 provided
makefile:14: recipe for target 'ShapeInherit.o' failed
make: *** [ShapeInherit.o] Error 1
```

```
ShapeInherit.cpp: In constructor 'Circle::Circle(float)':
ShapeInherit.cpp:28:3: error: no matching function for call to
'Shape::Shape()'
```

```
Shape(std::string name) : ShapeName{name}
{
}
```

C++ requires that a derived-class constructor (`Circle`) call its base class constructor (`Shape`) to initialize the base class (`Shape`) data members that are inherited into the derived class (`Circle`).

We could simply add default values to `Shape`'s constructor so that `Circle` can call `Shape`'s constructor without any parameters.

```
Shape(std::string name="BaseShape") : ShapeName{name}
{
}
```

But this solution does not allow us to name our `Circle` objects – they would all be `BaseShape`.

```
class Circle : public Shape
{
   public:
      Circle(float radius=0)
      {
         dim1 = dim2 = radius;
      }

      float getarea()
      {
         return dim1 * dim2 * M_PI;
      }
};
```

```
class Circle : public Shape
{
   public:
      Circle(std::string name, float radius=0)
      : Shape(name)
      {
         dim1 = dim2 = radius;
      }

       float getarea()
      {
         return dim1 * dim2 * M_PI;
      }
};
```

We need to alter `Circle`'s constructor – it needs to accept a name that it can then pass on to `Shape`'s constructor.

```
Shape A("Poly");

std::cout << "My name is " << A.getName() << std::endl;

Circle C("Hoop");

std::cout << "My name is " << C.getName() << std::endl;
```

```
My name is Poly
My name is Hoop
```

Object `C` of class `Cicle` is able to use the member function `getName()` even though class `Circle` does not contain `getName()`

`Cicle` inherited it from `Shape`

```
69          Shape A("Poly");

(gdb) ptype A

type = class Shape {
  public:
    float dim1;
    float dim2;
    std::__cxx11::string ShapeName;

    Shape(std::__cxx11::string);
    std::__cxx11::string getName(void);
}

(gdb) p A

$2 = {
  dim1 = 1.40129846e-45,
  dim2 = 0,
  ShapeName = "Poly"
}
```

```
73          Circle C("Hoop");

(gdb) ptype C

type = class Circle : public Shape {
  public:
    Circle(std::__cxx11::string, float);
    float getarea(void);
}

(gdb) p C

$3 = {
  <Shape> = {
    dim1 = 0,
    dim2 = 0,
    ShapeName = "Hoop"
  }, <No data fields>}
```

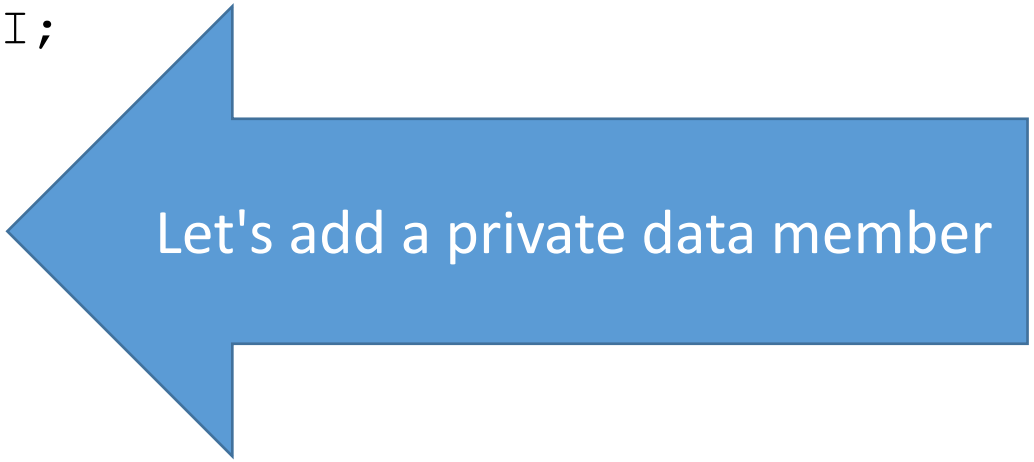
Why is `dim1` and `dim2` set to `0`?

```
Circle(std::string name, float radius=0)
: Shape(name)
{
   dim1 = dim2 = radius;
}
```

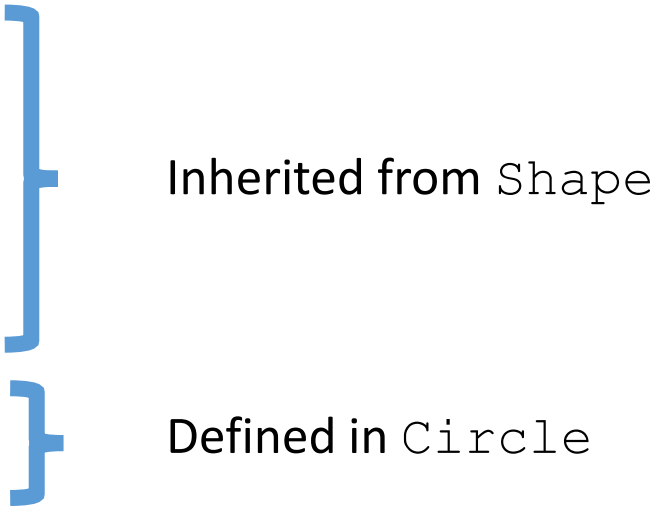
```
class Circle : public Shape
{
   public:
      Circle(std::string name, float radius=0)
      : Shape(name)
      {
         dim1 = dim2 = radius;
      }

      float getarea()
      {
         return dim1 * dim2 * M_PI;
      }

   private :
      std::string color;
};
```

And instantiate our object with a radius

```
Circle C("Hoop", 3);
```



```
76          Circle C("Hoop", 3);

(gdb) p C

$2 = {
  <Shape> = {
    dim1 = 3,
    dim2 = 3,
    ShapeName = "Hoop"
  },
  members of Circle:
  color = ""
}
```



```
class Circle : public Shape
{
   public:
      Circle(std::string name, float radius=0)
      : Shape(name)
      {
         dim1 = dim2 = radius;
      }

      float getarea()
      {
         return dim1 * dim2 * M_PI;
      }

   private :
      std::string color;
};
```

```
My name is Hoop and my area is 28.2743
```

```
Circle C("Hoop", 3);

std::cout << "My name is " << C.getName()
          << " and my area is "
          << C.getarea() << std::endl;
```

Let's add another derived class - Rectangle

```
class Rectangle : public Shape
{
  public:
    Rectangle(std::string name, float height=0, float width=0)
    : Shape(name)
    {
      dim1 = height;
      dim2 = width;
    }

    float getarea()
    {
      return dim1 * dim2;
    }
};
```

```
Rectangle R("NotQuiteSquare", 4, 6);

std::cout << "My name is " << R.getName()
          << " and my area is "
          << R.getarea() << std::endl;


My name is NotQuiteSquare and my area is 24
```

```
(gdb) p R
$2 = {
  <Shape> = {
    dim1 = 4,
    dim2 = 6,
    ShapeName = "NotQuiteSquare"
  }, <No data fields>}
```

Now, let's add a new shape – a square.

How is a square different from a rectangle?

A square is a special type of rectangle where all four sides have the same length.

So a square is a shape and a rectangle which means

- the area calculation for a square is the same as a rectangle.
- rectangle's area calculation requires two sides (l * w) so square could use the same calculation as long as length = width.

```
class Square
{
   public:
      Square(float size)
      {
         dim1 = size;
         dim2 = size;
      }
};
```

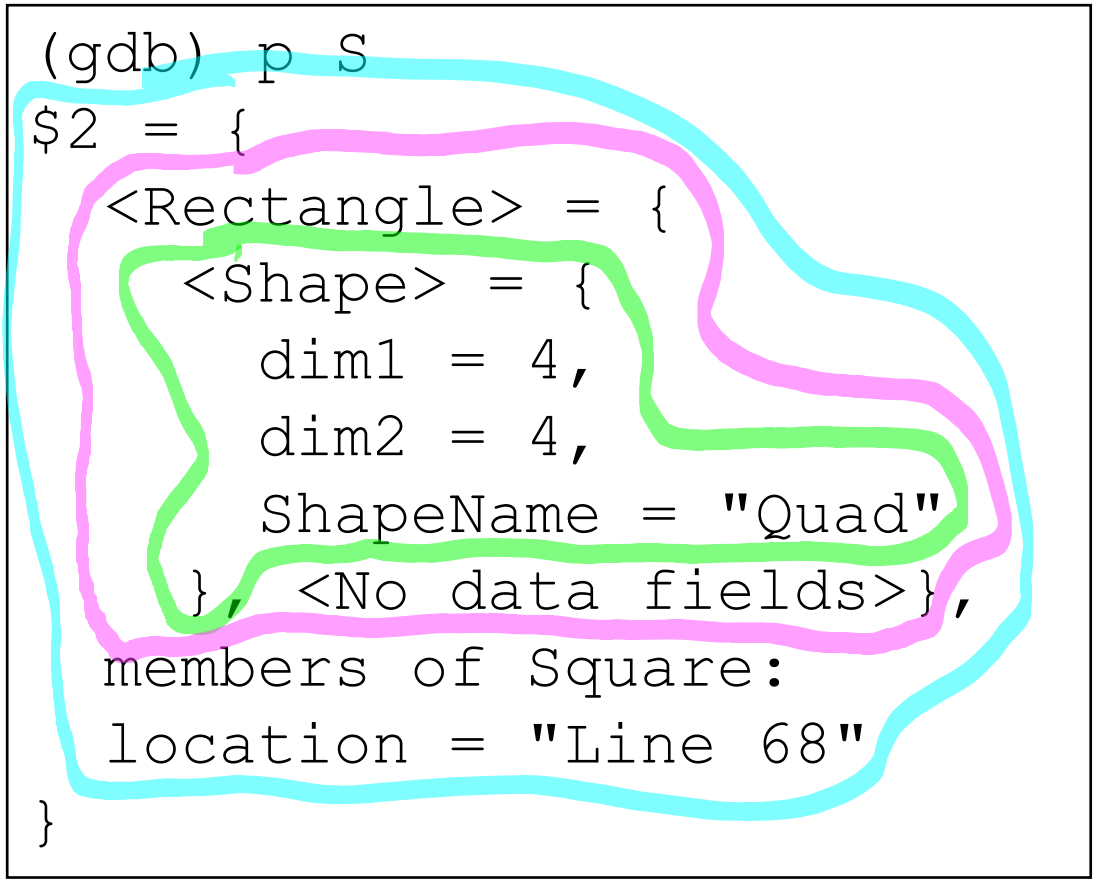
```
Square S("Quad", 4);

std::cout << "My name is " << S.getName()
          << " and my area is "
          << S.getarea() << std::endl;
```

```
My name is Quad and my area is 16
```

```
class Square : public Rectangle
{
   public:
      Square(std::string name, float size)
      : Rectangle(name, size)
      {
         dim1 = size;
         dim2 = size;
      }
};
```

```
85          Square S("Quad", 4);

(gdb) p S
$2 = {
  <Rectangle> = {
    <Shape> = {
      dim1 = 4,
      dim2 = 4,
      ShapeName = "Quad"
    }, <No data fields>}, <No data fields>}
```

```
class Square : public Rectangle
{
   public:
      Square(std::string name, float size)
      : Rectangle(name, size)
      {
         dim1 = size;
         dim2 = size;
      }
   private:
      std::string location{"Line 68"};
};
```



Inheritance

```
class Circle : public Shape

class Rectangle : public Shape

class Square : public Rectangle
```

colon (:) in the class definition indicates inheritance

keyword `public` indicates the type of inheritance

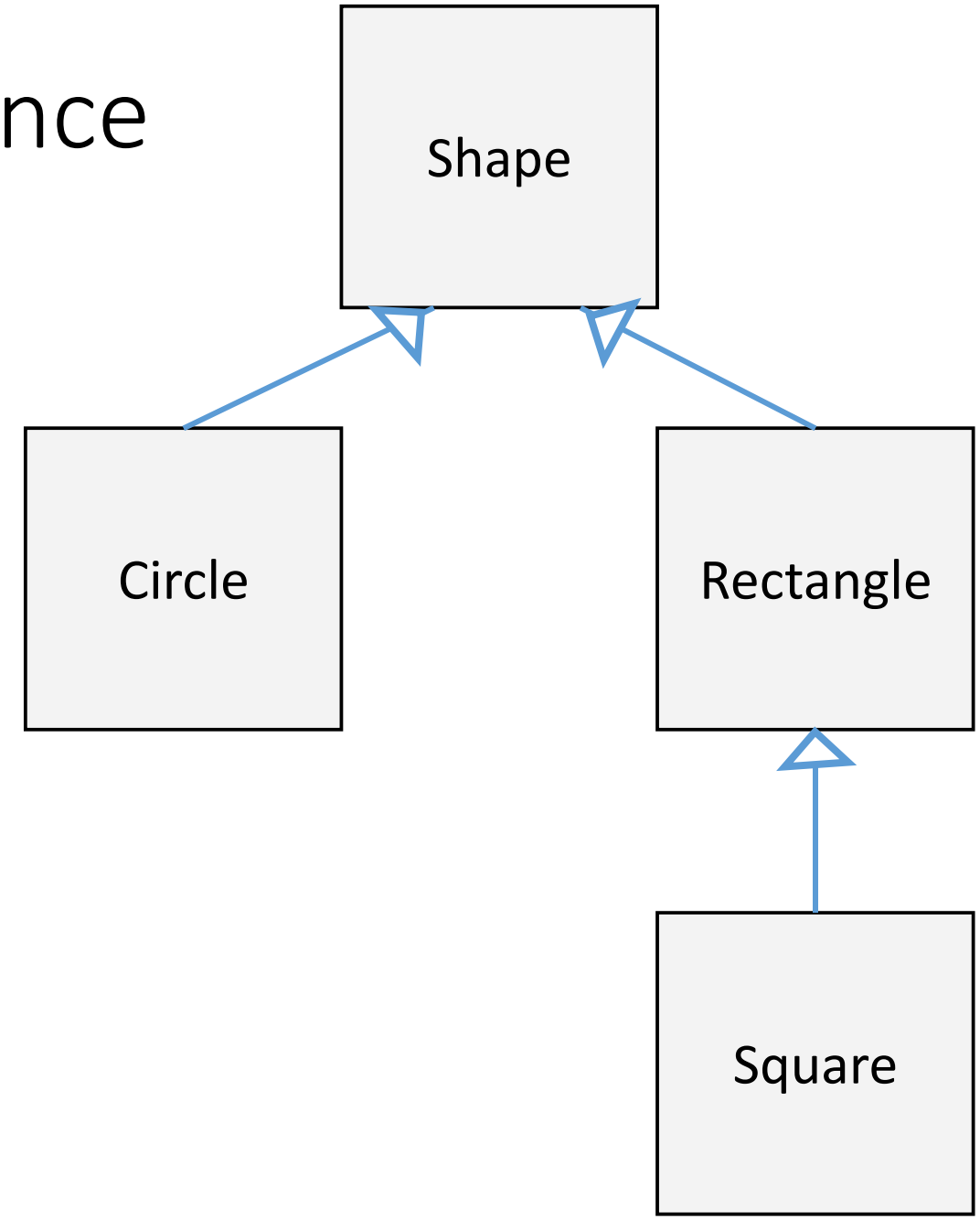
Constructors and destructors are not inherited

Inheritance

When C++ constructs derived objects, it does so in phases.

First, the most-base class (at the top of the inheritance tree) is constructed first.

Then each child class is constructed in order, until the most-child class (at the bottom of the inheritance tree) is constructed last.

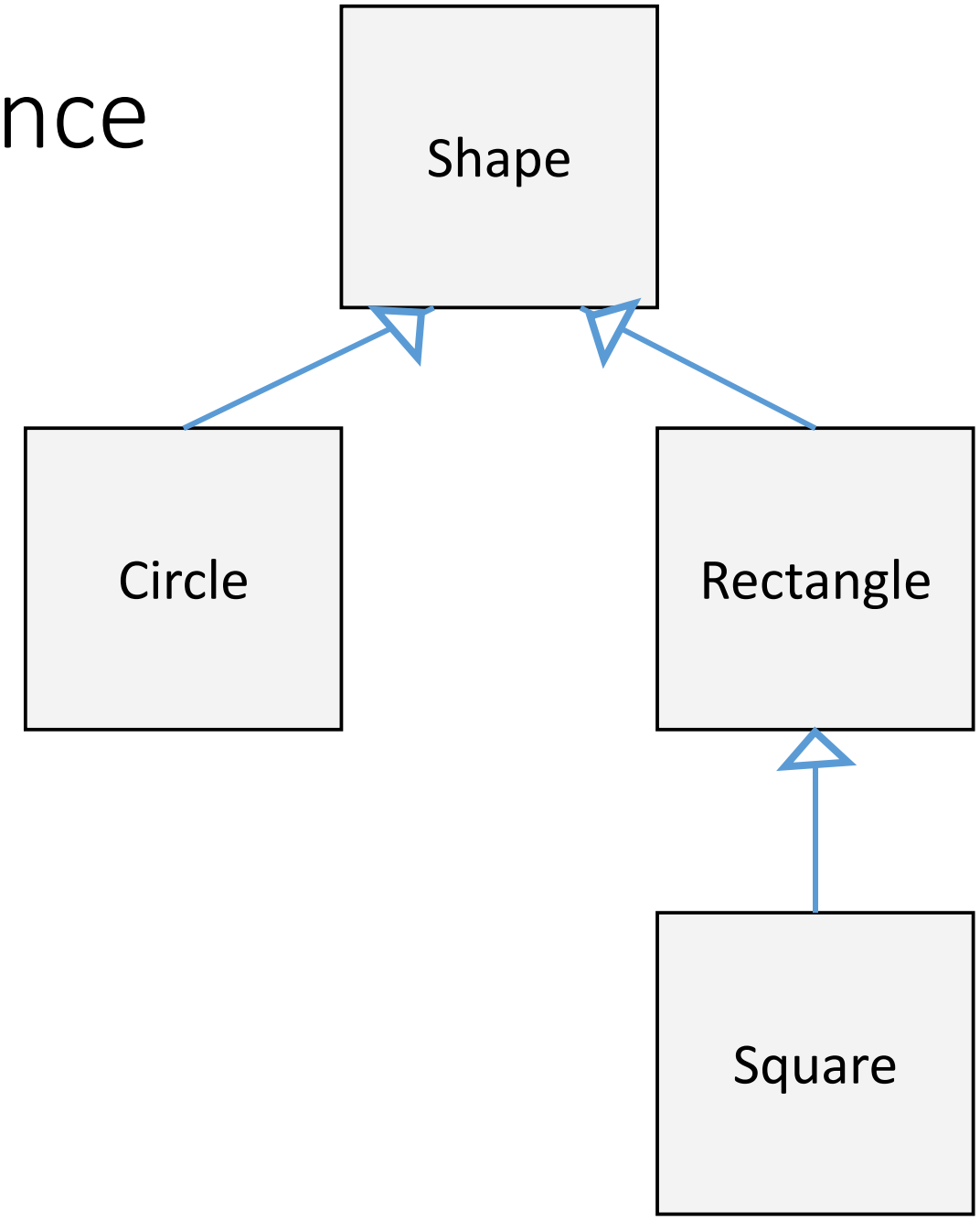


Inheritance

So when we construct a Circle, a Shape is constructed first and then the Circle is constructed.

When we construct a Square, a Shape is constructed and then a Rectangle and then a Square.

A child cannot exist until the parent exists.



Inheritance

```
std::cout << "Let's make a Shape" << std::endl;

Shape A("Poly");

std::cout << "My name is " << A.getName() << std::endl;
```

```
Let's make a Shape
SHAPE!
My name is Poly
```

```
Shape(std::string name="BaseShape")
: ShapeName{name}
{
   std::cout << "SHAPE!" << std::endl;
}
```

Inheritance

```
std::cout << "Let's make a Circle" << std::endl;

Circle C("Hoop", 3);

std::cout << "My name is " << C.getName()
          << " and my area is " << C.getarea()
          << std::endl;
```

```
Let's make a Circle
SHAPE!
CIRCLE!
My name is Hoop and my area is 28.2743
```

```
Circle(std::string name, float radius=0)
: Shape(name)
{
   dim1 = dim2 = radius;
   std::cout << "CIRCLE!" << std::endl;
}
```

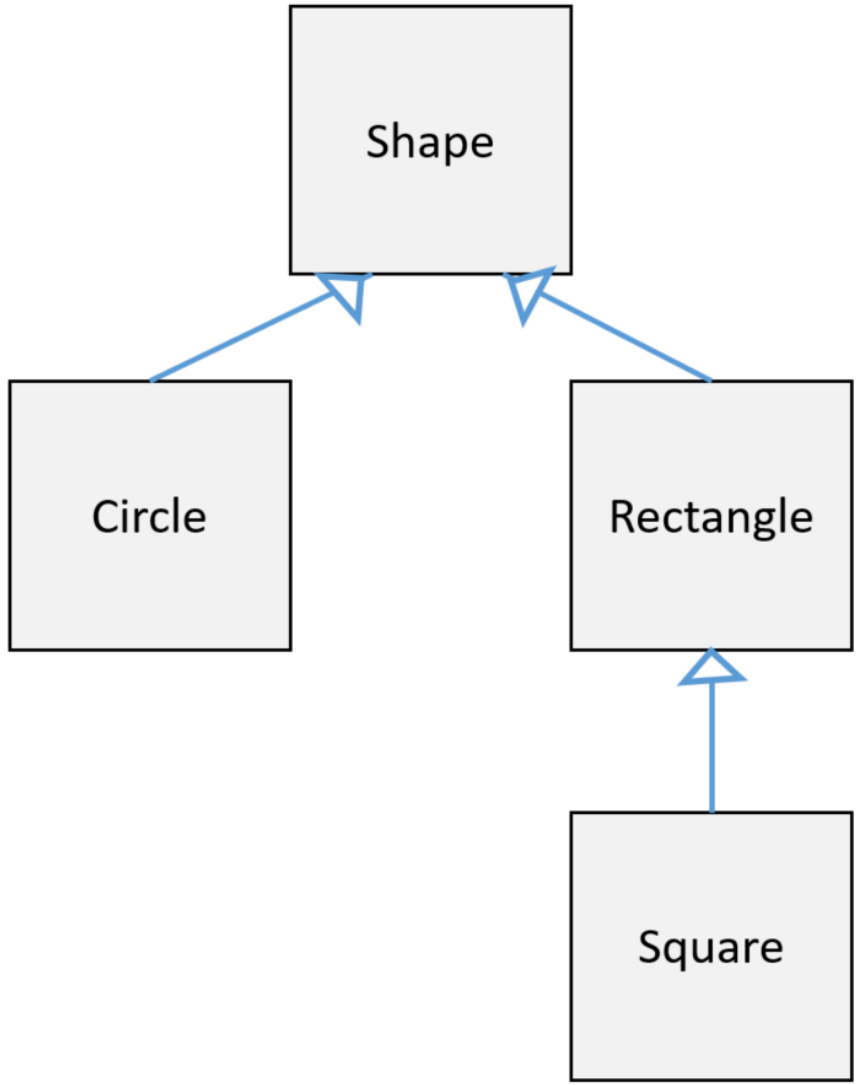
Inheritance

```
std::cout << "Let's make a Rectangle" << std::endl;

Rectangle R("NotQuiteSquare", 4, 6);

std::cout << "My name is " << R.getName()
          << " and my area is " << R.getarea()
          << std::endl;
```

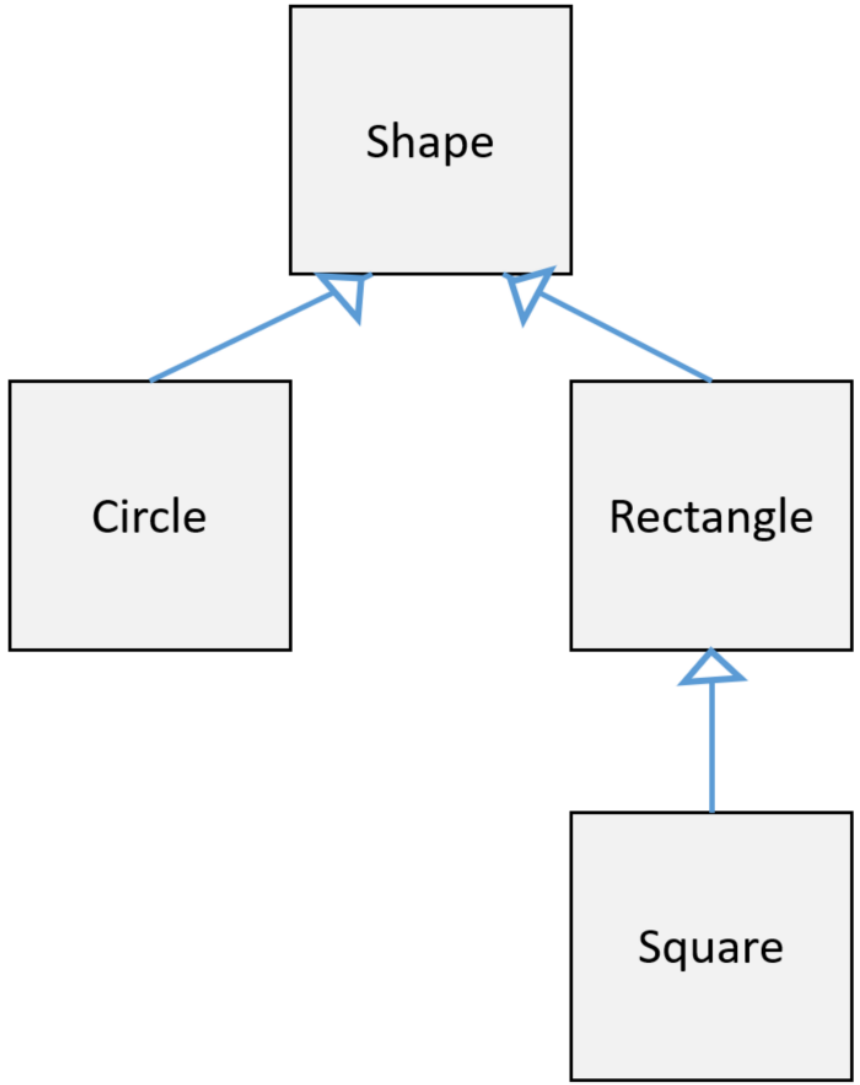
```
Let's make a Rectangle
SHAPE!
RECTANGLE!
My name is NotQuiteSquare and m
```

```
Rectangle(std::string name, float
height=0, float width=0)
: Shape(name)
{
   dim1 = height;
   dim2 = width;
   std::cout << "RECTANGLE!"
             << std::endl;
}
```

```
Square(std::string name, float size)
: Rectangle(name, size)
{
      dim1 = size;
      dim2 = size;
      std::cout << "SQUARE!" << std::endl;

}

std::cout << "Let's make a Square" << std::endl;

Square S("Quad", 4);

std::cout << "My name is " << S.getName()
          << " and my area is " << S.getarea()
          << std::endl;
```



```
Let's make a Square
SHAPE!
RECTANGLE!
SQUARE!
My name is Quad and my area is 16
```



Inheritance

The derived class often uses variables and functions from the base class but the base class knows nothing about the derived class.

Instantiating the base class first ensures those variables are already initialized by the time the derived class is created and ready to use them.

Remember that C++ always constructs the “first” or “most base” class first. It then walks through the inheritance tree in order and constructs each successive derived class.

Inheritance

With non-derived classes, constructors only have to worry about their own members.

For example, consider `Shape`. We can create a `Shape` object like this:

```
Shape A("Poly");
```

Here's what actually happens when `Shape` is instantiated:

- Memory for `Shape` is set aside
- The appropriate `Shape` constructor is called
- The initialization list initializes variables
- The body of the constructor executes
- Control is returned to the caller

Inheritance

With derived classes, a few more things happen

For example, consider `Circle`. We can create a `Circle` object like this:

```
Circle A("Hoop");
```

Here's what actually happens when `Circle` is instantiated:

- Memory for derived is set aside (enough for both the `Shape` and `Circle` portions)
- The appropriate `Circle` constructor is called
- The `Shape` object is constructed first using the appropriate `Shape` constructor. If no base constructor is specified, the default constructor will be used.
- The initialization list initializes variables
- The body of the constructor executes
- Control is returned to the caller

Inheritance

The only real difference between constructing an object that inherits and an object that does not inherit is that before the Derived constructor can do anything substantial, the Base constructor is called first.

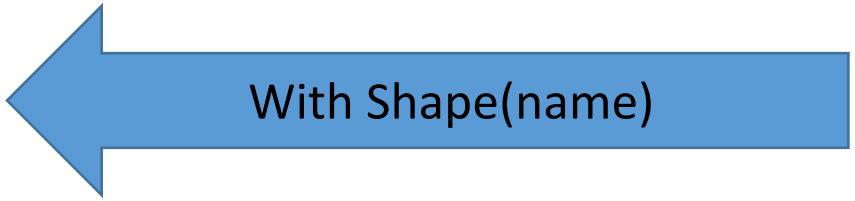
The Base constructor sets up the Base portion of the object, control is returned to the Derived constructor, and the Derived constructor is allowed to finish up its job.

```
class Circle : public Shape
{
   public:
      Circle(std::str
      : Shape(name)
      {
         dim1 = dim2 = r  ius;
         std::cout << "CIRCLE!" << std::endl;
      }

      float getarea()
      {
         return dim1 * dim2 * M_PI;
      }

   private :
      std::string color;
};
```

What happens if we call `Shape`'s constructor as `Shape()` rather than `Shape(name)`?

```
Shape(std::string name="BaseShape") :
ShapeName{name}
{
   std::cout << "SHAPE!" << std::endl;
}
```

```
Let's make a Circle
SHAPE!
CIRCLE!
My name is Hoop and my area is 28.2743
```




```
Let's make a Circle
SHAPE!
CIRCLE!
My name is BaseShape and my area is 28.2743
```

We called `Shape()` and not `Shape(name)` so the default parameter value of `BaseShape` was used to construct `Circle`.

```
Shape(std::string name="BaseShape") :
ShapeName{name}
{
    std::cout << "SHAPE!" << std::endl;
}
```

Constructors and Destructors in Derived Classes

- Instantiating a derived-class object begins a *chain* of constructor calls in which the derived-class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class's default constructor).

- If the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on.

- The last constructor called in this chain is the constructor of the class at the base of the hierarchy, whose body actually finishes executing *first*.

- The most derived-class constructor's body finishes executing *last*.

- Each base-class constructor initializes the base-class data members that the derived-class object inherits.

Constructors and Destructors in Derived Classes

- When a derived-class object is destroyed, the program calls that object's destructor.
- This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in *reverse* of the order in which the constructors executed.
- When a derived-class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy.
- This process repeats until the destructor of the final base class at the top of the hierarchy is called.
- Then the object is removed from memory.

Constructors and Destructors in Derived Classes

Base-class constructors, destructors and overloaded assignment operators are *not* inherited by derived classes.

Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class versions.

If the derived class does not explicitly define constructors, the compiler still generates a default constructor in the derived class.

```
class Shape
{
  public :
    Shape(std::string name="BaseShape") : ShapeName{name}
    {
      std::cout << "SHAPE!" << std::endl;
    }
    std::string getName()
    {
      return ShapeName;
    }



    float dim1;
    float dim2;
    std::string ShapeName;
};
```

Up to now, we kept `Shape`'s data members in the public access. What happens if we change them to private?

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11  ShapeInheritp.cpp -o ShapeInheritp.o
ShapeInheritp.cpp: In constructor ‘Circle::Circle(std::__cxx11::string, float)’:
ShapeInheritp.cpp:33:4: error: ‘float Shape::dim1’ is private within this context
    dim1 = dim2 = radius;
    ^~~
ShapeInheritp.cpp:22:9: note: declared private here
   float dim1;
         ^~~
ShapeInheritp.cpp:33:11: error: ‘float Shape::dim2’ is private within this context
    dim1 = dim2 = radius;
           ^~~
ShapeInheritp.cpp:23:9: note: declared private here
   float dim2;
         ^~~
ShapeInheritp.cpp: In member function ‘float Circle::getarea()’:
ShapeInheritp.cpp:39:11: error: ‘float Shape::dim1’ is private within this context
    return dim1 * dim2 * M_PI;
           ^~~
ShapeInheritp.cpp:22:9: note: declared private here
   float dim1;
         ^~~
ShapeInheritp.cpp:39:18: error: ‘float Shape::dim2’ is private within this context
    return dim1 * dim2 * M_PI;
                 ^~~
ShapeInheritp.cpp:23:9: note: declared private here
   float dim2;
         ^~~
ShapeInheritp.cpp: In constructor ‘Rectangle::Rectangle(std::__cxx11::string, float, float)’:
ShapeInheritp.cpp:52:4: error: ‘float Shape::dim1’ is private within this context
    dim1 = height;
    ^~~
ShapeInheritp.cpp:22:9: note: declared private here
   float dim1;
         ^~~
ShapeInheritp.cpp:53:4: error: ‘float Shape::dim2’ is private within this context
    dim2 = width;
    ^~~
ShapeInheritp.cpp:23:9: note: declared private here
   float dim2;
         ^~~
ShapeInheritp.cpp: In member function ‘float Rectangle::getarea()’:
ShapeInheritp.cpp:59:11: error: ‘float Shape::dim1’ is private within this context
    return dim1 * dim2;
           ^~~
ShapeInheritp.cpp:22:9: note: declared private here
   float dim1;
         ^~~
ShapeInheritp.cpp:59:18: error: ‘float Shape::dim2’ is private within this context
    return dim1 * dim2;
                 ^~~
ShapeInheritp.cpp:23:9: note: declared private here
   float dim2;
         ^~~
ShapeInheritp.cpp: In constructor ‘Square::Square(std::__cxx11::string, float)’:
ShapeInheritp.cpp:69:4: error: ‘float Shape::dim1’ is private within this context
    dim1 = size;
    ^~~
ShapeInheritp.cpp:22:9: note: declared private here
   float dim1;
         ^~~
ShapeInheritp.cpp:70:4: error: ‘float Shape::dim2’ is private within this context
    dim2 = size;
    ^~~
ShapeInheritp.cpp:23:9: note: declared private here
   float dim2;
         ^~~
makefile:14: recipe for target 'ShapeInheritp.o' failed
make: *** [ShapeInheritp.o] Error 1
```

```
ShapeInheritp.cpp: In constructor ‘Circle::Circle(std::__cxx11::string,
float)’:
ShapeInheritp.cpp:33:4: error: ‘float Shape::dim1’ is private within this
context
    dim1 = dim2 = radius;
    ^~~~
ShapeInheritp.cpp:22:9: note: declared private here
   float dim1;
         ^~~~
ShapeInheritp.cpp:33:11: error: ‘float Shape::dim2’ is private within this
context
    dim1 = dim2 = radius;
           ^~~~
ShapeInheritp.cpp:23:9: note: declared private here
   float dim2;
```

Inheritance

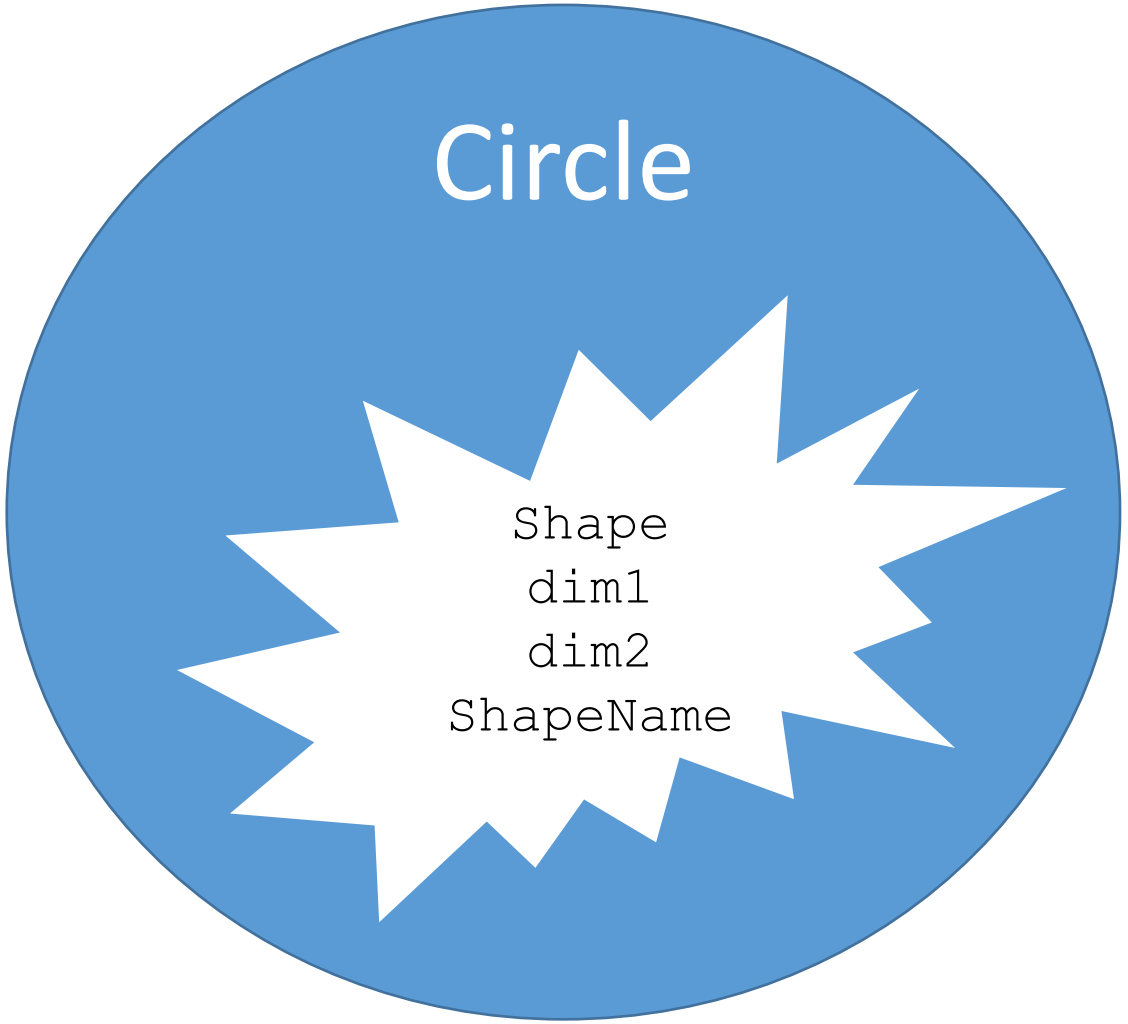
Public members can be accessed by anybody.

Private members can only be accessed by member functions of the same class or friends.

This means derived classes cannot access private members of the base class directly.

Derived class `Circle` inherits `Shape`'s information but is not allowed to directly access the private data members inherited from `Shape`.

Derived classes will need to use access functions to access private members of the base class.



We would need to add getters and setters to `Shape` to provide access to private data members.

```
void set_dims(float Dim1, float Dim2)
{
     dim1 = Dim1;
     dim2 = Dim2;
}
float get_dim1()
{
     return dim1;
}
float get_dim2()
{
     return dim2;
}
```

We would then change `Circle` to use them.

```
Circle(std::string name, float radius=0)
: Shape(name)
{
    set_dims(radius, radius);
    [redacted]
    std::cout << "CIRCLE!" << std::endl;
}

float getarea()
{
    [redacted]
    return get_dim1() * get_dim2() * M_PI;
}
```

```
Let's make a Circle
SHAPE!
CIRCLE!
My name is Hoop and my area is 28.2743
```

Including the Base-Class Header in the Derived-Class Header with `#include`

We `#include` the base class's header in the derived class's header. This is necessary for three reasons.

The derived class uses the base class's name, so we must tell the compiler that the base class exists.

The compiler uses a class definition to determine the size of an object of that class. A client program that creates an object of a class `#include`s the class definition to enable the compiler to reserve the proper amount of memory.

The compiler must determine whether the derived class uses the base class's inherited members properly.

```
class Common
{
	public :
		void getInfo(void)
		{
			cout << "Enter your name " << endl;
			cin >> name;
			cout << "Enter your gender " << endl;
			cin >> gender;
			cout << "\nEnter your age " << endl;
			cin >> age;
		}
		void displayInfo(void)
		{
			cout << "Display Data" << endl;
			cout << "Name\t" << name << endl;
			cout << "Gender\t" << gender << endl;
			cout << "Age\t" << age << endl;
		}
	private :
		string name;
		string gender;
		int age;
};
```

```
class Principal : public Common
{
  public :
    void getSalary(void)
    {
      cout << "Enter Principal salary ";
      cin >> salary;
    }

    void showSalary(void)
    {
      cout << "Principal Salary : "
           << salary << endl;
    }
  private :
    int salary;
};
```

```
87              Principal Pal;

(gdb) p Pal
$2 = {
  <Common> = {
    name = "",
    gender = "",
    age = 6299808
  },
  members of Principal:
  salary = 0
}
```

```
class Teacher : public Common
{
  public :
    void getSalary(void)
    {
      cout << "Enter Teacher salary ";
      cin >> salary;
    }

    void showSalary(void)
    {
      cout << "Teacher Salary : "
           << salary << endl;
    }
  private :
    int salary;
};
```

```
94              Teacher Mr;


(gdb) p Mr
$3 = {
  <Common> = {
    name = "",
    gender = "",
    age = 6299112
  },
  members of Teacher:
  salary = 0
}
```

```
class Student : public Common
{
  public :
    void getGrade(void)
    {
      cout << "Enter Student grade ";
      cin >> grade;
    }

    void showGrade(void)
    {
      cout << "Student Grade : "
           << grade << endl;
    }
  private :
    int grade;
};
```

```
101          Student You;

(gdb) p You
$4 = {
  <Common> = {
    name = "",
    gender = "",
    age = 4198944
  },
  members of Student:
  grade = 0
}
```

```
87    Principal Pal;
(gdb) n
88    Pal.getInfo();
(gdb) n
Enter your name
Fred

Enter your gender
Male

Enter your age
34
```

```
89    Pal.getSalary();
(gdb) n
Enter Principal salary 12345
91    Pal.displayInfo();
(gdb) n
Display Data
Name  Fred
Gender Male
Age 34
92    Pal.showSalary();
(gdb) n
Principal Salary : 12345
```

```
(gdb) p Pal
$5 = {
  <Common> = {
    name = "Fred",
    gender = "Male",
    age = 34
  },
  members of Principal:
  salary = 12345
}
(gdb) ptype Pal
type = class Principal : public
Common {
  private:
    int salary;

  public:
    void getSalary(void);
    void showSalary(void);
}
```

```
94    Teacher Mr;
(gdb) n
95    Mr.getInfo();
(gdb) n
Enter your name
Bob

Enter your gender
Male

Enter your age
45
```

```
96    Mr.getSalary();
(gdb) n
Enter Teacher salary 23456
98    Mr.displayInfo();
(gdb) n
Display Data
Name  Bob
Gender Male
Age 45
99    Mr.showSalary();
(gdb) n
Principal Salary : 23456
```

```
(gdb) p Mr
$6 = {
  <Common> = {
    name = "Bob",
    gender = "Male",
    age = 45
  },
  members of Teacher:
  salary = 23456
}
(gdb) ptype Mr
type = class Teacher : public
Common {
  private:
    int salary;

  public:
    void getSalary(void);
    void showSalary(void);
}
```

```
101   Student You;
(gdb) n
102   You.getInfo();
(gdb) n
Enter your name
Mary

Enter your gender
Female

Enter your age
12
```

```
103   You.displayInfo();
(gdb) n
Display Data
Name  Mary
Gender  Female
Age 12
104   You.getGrade();
(gdb) n
Enter Student grade A
105   You.showGrade();
(gdb) n
Student Grade : 0
```

```
(gdb) p You
$2 = {
  <Common> = {
    name = "Mary",
    gender = "Female",
    age = 12
  },
  members of Student:
  grade = 0
}
(gdb) ptype You
type = class Student : public
Common {
  private:
    int grade;

  public:
    void getGrade(void);
    void showGrade(void);
}
```

`protected` Data

To enable class `Rectangle` to directly access `Shape` data members `dim1` and `dim2`, we can declare those members as `protect`ed in the base class.

A base class's `protected` members can be

accessed within the body of that base class,

by members and friends of that base class, and

by members and friends of any classes derived from that base class.

```
class Shape
{
  public :
    Shape(std::string name="BaseShape") : ShapeName{name}
    {
      std::cout << "SHAPE!" << std::endl;
    }
    std::string getName()
    {
      return ShapeName;
    }

  protected :

    float dim1;
    float dim2;
    std::string ShapeName;
};
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11  ShapeInheritp.cpp -o
ShapeInheritp.o
g++ -g -std=c++11  ShapeInheritp.o -o ShapeInheritp.e
student@cse1325:/media/sf_VM$ ./ShapeInheritp.e
Let's make a Shape
SHAPE!
My name is Poly
Let's make a Circle
SHAPE!
CIRCLE!
My name is Hoop and my area is 28.2743
Let's make a Rectangle
SHAPE!
RECTANGLE!
My name is NotQuiteSquare and my area is 24
Let's make a Square
SHAPE!
RECTANGLE!
SQUARE!
My name is Quad and my area is 16
```

`protected` Data

- `Rectangle` and `Circle` inherit from class `Shape`.

- Objects of class `Rectangle` and `Circle` can access inherited data members that are declared `protected` in class `Shape`.

- Objects of a derived class also can access `protected` members in any of that derived class's *indirect* base classes.

- `Square` inherits from `Rectangle` which inherits from `Shape` and `Square` can access `Shape`'s `protected` members.

Notes on Using `protected` Data

Inheriting `protected` data members slightly increases performance because we can directly access the members without incurring the overhead of calls to *set* or *get* member functions.

In most cases, it's better to use `private` data members to encourage proper software engineering and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.

`protected` data members are notated in class diagrams with the # mark

Notes on Using `protected` Data

Using `protected` data members creates two serious problems.

1. The derived-class object does not have to use a member function to set the value of the base class's protected data member.

Our private data member `color` is set to private and is kept from being set to the value of "green" by the setter for it.


```
void setColor(std::string Color)
{
   if (Color == "green")
   {
      throw std::invalid_argument("No green!!");
   }
   color = Color;
}

private :
   std::string color;
```

```
Square S("Quad", 4);

S.setColor("green");
```


```
Let's make a Square
SHAPE!
RECTANGLE!
SQUARE!
terminate called after throwing an instance of
'std::invalid_argument'
  what():  No green!!
Aborted (core dumped)
```

If we create a function in class `Square` to set the inherited private data member `color`…

```
void setSquareColor(std::string Color)
{
      color = Color;
}
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11  ShapeInheritp.cpp -o ShapeInheritp.o
ShapeInheritp.cpp: In member function ‘void
Square::setSquareColor(std::__cxx11::string)’:
ShapeInheritp.cpp:101:4: error: ‘std::__cxx11::string Rectangle::color’ is
private within this context
    color = Color;
    ^~~~~
ShapeInheritp.cpp:86:15: note: declared private here
   std::string color;
               ^~~~~
makefile:14: recipe for target 'ShapeInheritp.o' failed
make: *** [ShapeInheritp.o] Error 1
```

If we change our `Rectangle` private data member `color` to protected.

From

```
private :
    std::string color;
```

To

```
protected :
    std::string color;
```

```
129          S.setSquareColor("green");
(gdb) s
Square::setSquareColor (this=0x7fffffffdfd0, Color="green") at
ShapeInheritp.cpp:101
101                       color = Color;

(gdb) p S
$2 = {
  <Rectangle> = {
    <Shape> = {
      dim1 = 4,
      dim2 = 4,
      ShapeName = "Quad"
    },
    members of Rectangle:
    color = "green"
  },
  members of Square:
  location = "Line 68"
}
```

Inherited data member `color` was set to "`green`" even though `Rectangle` has a setter for `color` that explicitly does not allow the value of "`green`" for `color`.

Notes on Using `protected` Data

Using `protected` data members creates two serious problems.

2. Derived-class member functions are more likely to be written so that they depend on the base-class implementation.

Derived classes should depend only on the base-class services (i.e., non-private member functions) and not on the base-class implementation.

- With `protected` data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class.
- Such software is said to be fragile or brittle, because a small change in the base class can "break" derived-class implementation.

`public`, `protected` and `private` Inheritance

When deriving a class from a base class, the base class may be inherited through `public`, `protected` or `private` inheritance.

Use of `protected` and `private` inheritance is rare.

A base class's `private` members are *never* accessible directly from a derived class, but can be accessed through calls to the `public` and `protected` members of the base class.

`public`, `protected` and `private` Inheritance

If you do not choose an inheritance type, C++ defaults to private inheritance (just like members default to private access if you do not specify otherwise).

There are 9 combinations: 3 member access specifiers (public, private, and protected), and 3 inheritance types (public, private, and protected).

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

`public`, `protected` and `private` Inheritance

So what's the difference between these?

When members are inherited, the access specifier for an inherited member may be changed (in the derived class only) depending on the type of inheritance used.

Members that were public or protected in the base class may change access specifiers in the derived class.

`public`, `protected` and `private` Inheritance

A class (and friends) can always access its own non-inherited members.

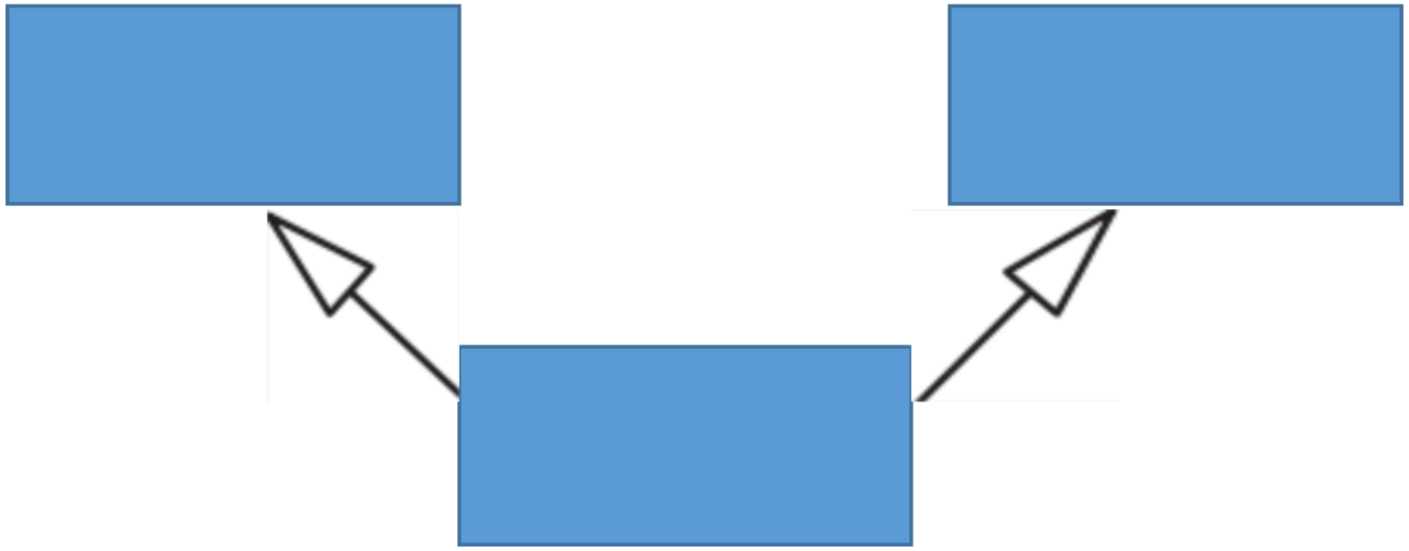
The access specifiers only affect whether outsiders and derived classes can access those members.

When derived classes inherit members, those members may change access specifiers in the derived class.

This does not affect the derived classes' own (non-inherited) members (which have their own access specifiers). It only affects whether outsiders and classes derived from the derived class can access those inherited members.

Multiple Inheritance

Multiple Inheritance occurs when a derived class inherits the members of two or more base classes.



Multiple Inheritance

Multiple Inheritance is a powerful capability that encourages interesting forms of software reuse.

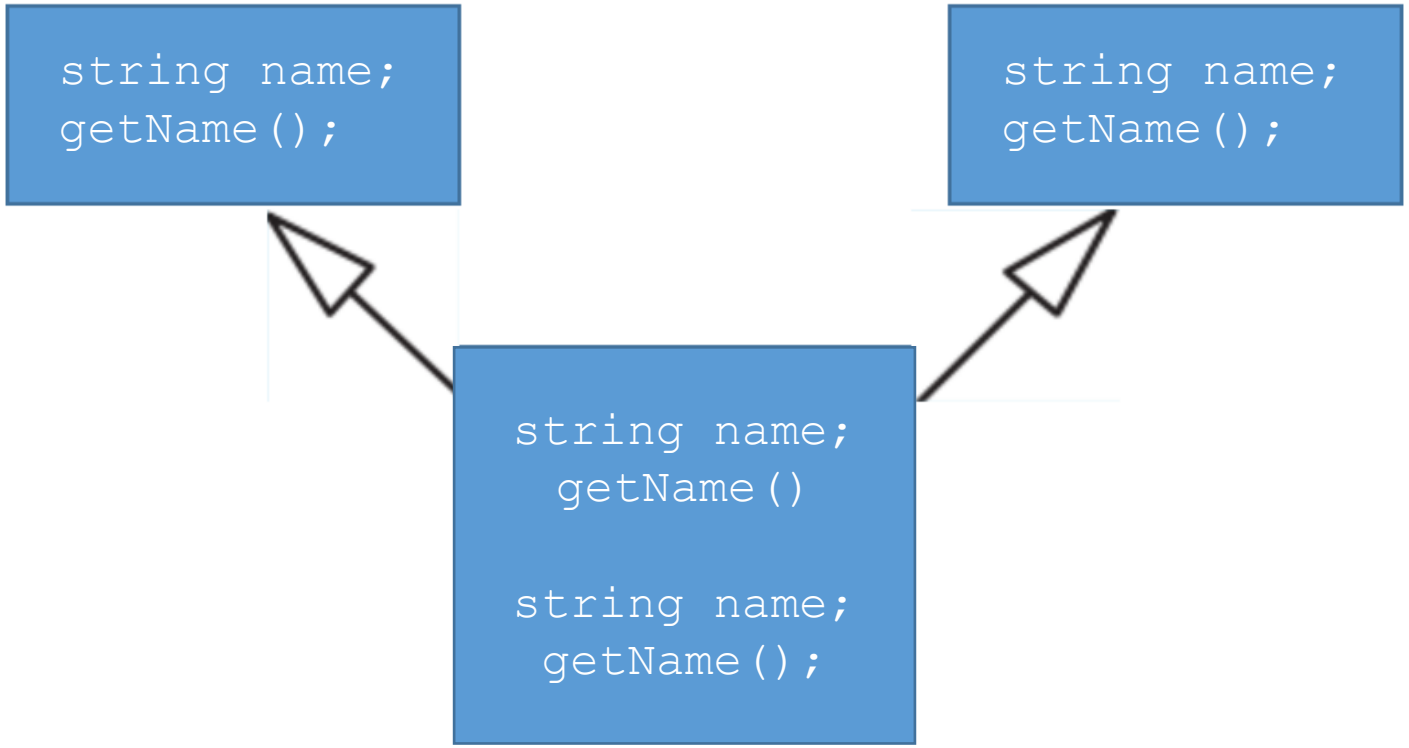
It can also cause a variety of ambiguity problems.

It is a difficult concept that should be used only be experienced programmers.

Some of the problems associated with multiple inheritance are so subtle that newer programming languages such as Java and C# do not enable a class to derive from more than one base class.

Multiple Inheritance

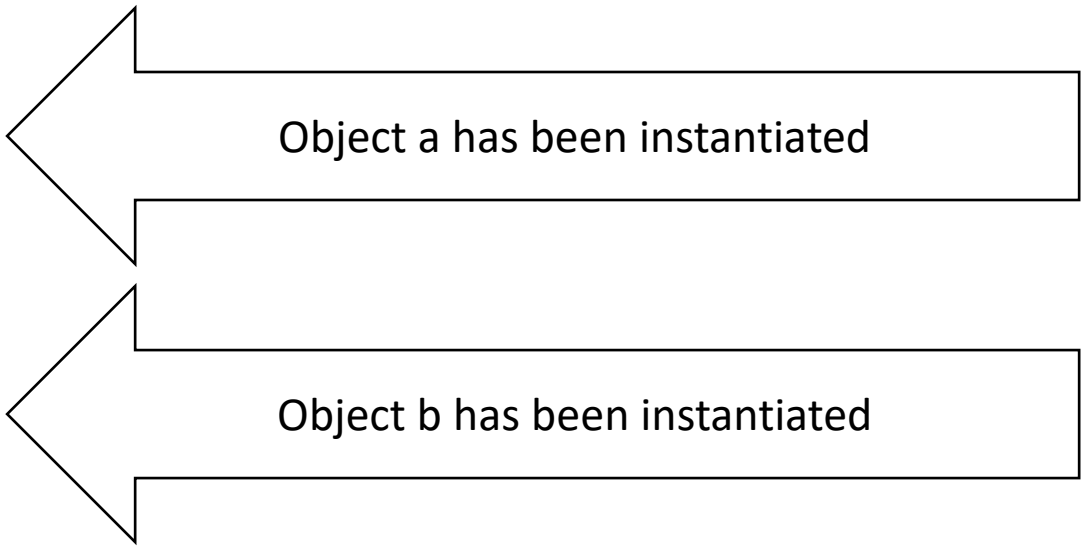
Multiple Inheritance can cause a situation where the derived class inherits data members or member functions from base classes that share names.



```
class A
{
   public :
      A(string myname="")
      {
         name = myname;
      }
      string name;
};

class B
{
   public :
      B(string myname="")
      {
         name = myname;
      }
      string name;
};
```

```
class AB : public A, public B
{
   public :
      string ABVar;
};

int main(void)
{
   A a("IamA");
   B b("IamB");
   AB ab;

   return 0;
}
```

```
35              A a("IamA");
(gdb) n
36              B b("IamB");
(gdb) n

(gdb) p a
$6 = {
  name = "IamA"
}
(gdb) p b
$7 = {
  name = "IamB"
}
```



```
37          AB ab;
(gdb) step
AB::AB (this=0x7fffffffe090) at
miDemo.cpp:27
27    class AB : public A, public B

A::A (this=0x7fffffffe090, myname="") at
miDemo.cpp:11
11                  {
(gdb)
12                      name = myname;
(gdb)
13                  }

B::B (this=0x7fffffffe0b0, myname="") at
miDemo.cpp:21
21                  {
(gdb)
22                      name = myname;
(gdb)
23                  }
```

constructor for A

constructor for B

```
(gdb) p ab
$8 = {
  <A> = {
    name = ""
  },
  <B> = {
    name = ""
  },
  members of
AB:
  ABVar = ""
}
```

Multiple Inheritance

The base class constructors are called in the order of that the inheritance is specified – not in the order in which their constructors are mentioned.

Constructor for A was called before constructor for B because AB inherited A before B

```
class AB : public A, public B
```

If the base class constructors are not explicitly called in the member initializer list, their default constructors are called implicitly.

```
37          AB ab;
(gdb) step
AB::AB (this=0x7fffffffe090) at miDemo.cpp:27
27      class AB : public B, public A
(gdb)
B::B (this=0x7fffffffe090, myname="") at miDemo.cpp:21
21              {
(gdb)
22                      name = myname;
(gdb)
23              }
(gdb)
A::A (this=0x7fffffffe0b0, myname="") at miDemo.cpp:11
11              {
(gdb)
12                      name = myname;
(gdb)
13              }
```

Multiple Inheritance

```
int main(void)
{
	A a("IamA");
	B b("IamB");
	AB ab;

	cout << "Object A's name is " << a.name << endl;
	cout << "Object B's name is " << b.name << endl;

	return 0;
}
```

```
Object A's name is IamA
Object B's name is IamB
```

Multiple Inheritance

```
int main(void)
{
    A a("IamA");
    B b("IamB");
    AB ab;

    cout << "Object A's name is " << a.name << endl;
    cout << "Object B's name is " << b.name << endl;
    cout << "Object AB's name is " << ab.name << endl;

    return 0;
}
```

So what happens when we try to print AB's name?

```
(gdb) p ab
$8 = {
  <A> = {
    name = ""
  },
  <B> = {
    name = ""
  },
  members of
AB:
  ABVar = ""
}
```

```
miDemo.cpp: In function 'int main()':
miDemo.cpp:41:39: error: request for member 'name' is ambiguous
  cout << "Object AB's name is " << ab.name << endl;
                                       ^

miDemo.cpp:14:10: note: candidates are: std::__cxx11::string A::name
   string name;
          ^

miDemo.cpp:24:10: note:                 std::__cxx11::string B::name
   string name;
          ^
```

Multiple Inheritance

We run into the same issue if we try to create a constructor for AB.

```
class AB : public B, public A
{
   public :
      AB(string myname="")
      {
         name = myname;
      }
      string ABVar;
};
```

```
miDemo.cpp: In constructor 'AB::AB(std::__cxx11::string)':
miDemo.cpp:32:4: error: reference to 'name' is ambiguous
    name = myname;
    ^
miDemo.cpp:14:10: note: candidates are: std::__cxx11::string A::name
   string name;
          ^
miDemo.cpp:24:10: note:                 std::__cxx11::string B::name
   string name;
          ^
```

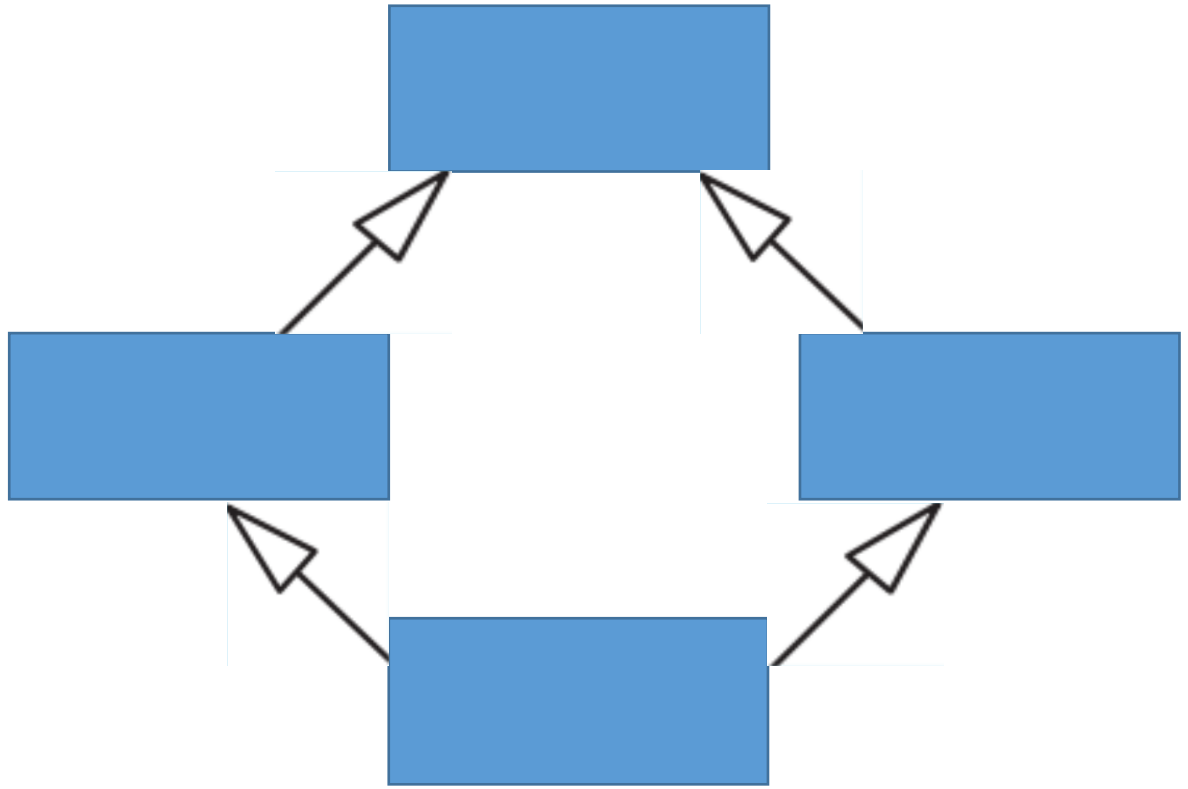
```
class AB : public B, public A
{
    public :
        AB(string myname="")
        {
            A::name ="A"+myname;
            B::name = "B"+myname;
        }
        string ABVar;
};

int main(void)
{
    A a("IamA");
    B b("IamB");
    AB ab("IamAB");

    cout << "Object A's name is " << a.name << endl;
    cout << "Object B's name is " << b.name << endl;
    cout << "Object AB's A name is " << ab.A::name << endl;
    cout << "Object AB's B name is " << ab.B::name << endl;

    return 0;
}
```

```
Object A's name is IamA
Object B's name is IamB
Object AB's A name is AIamAB
Object AB's B name is BIamAB
```

Multiple Inheritance
Diamond Inheritance

Diamond Inheritance occurs when a derived class inherits the members of two or more base classes who themselves inherited from a single base class.

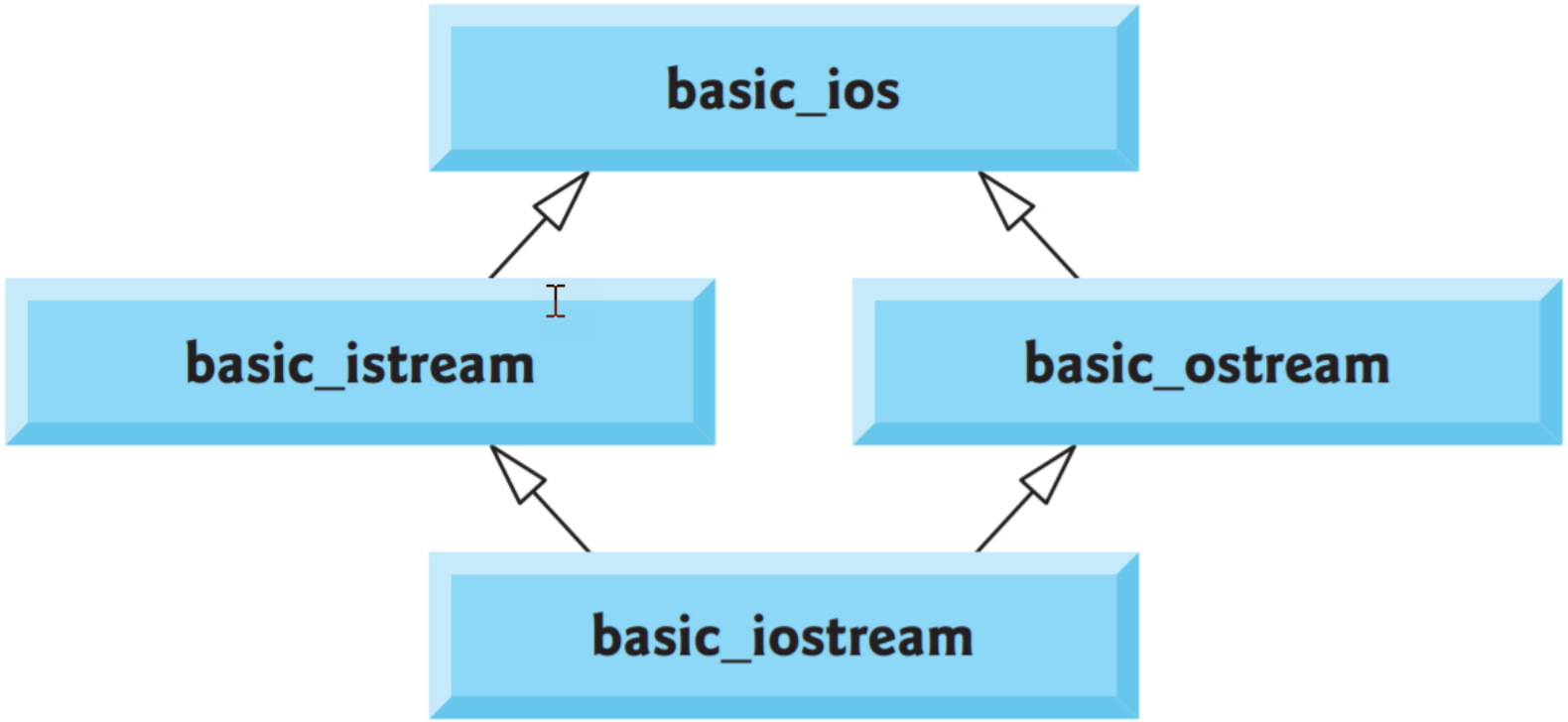
Not limited to 2 in the middle layer



Just need 1 on top and 1 on the bottom

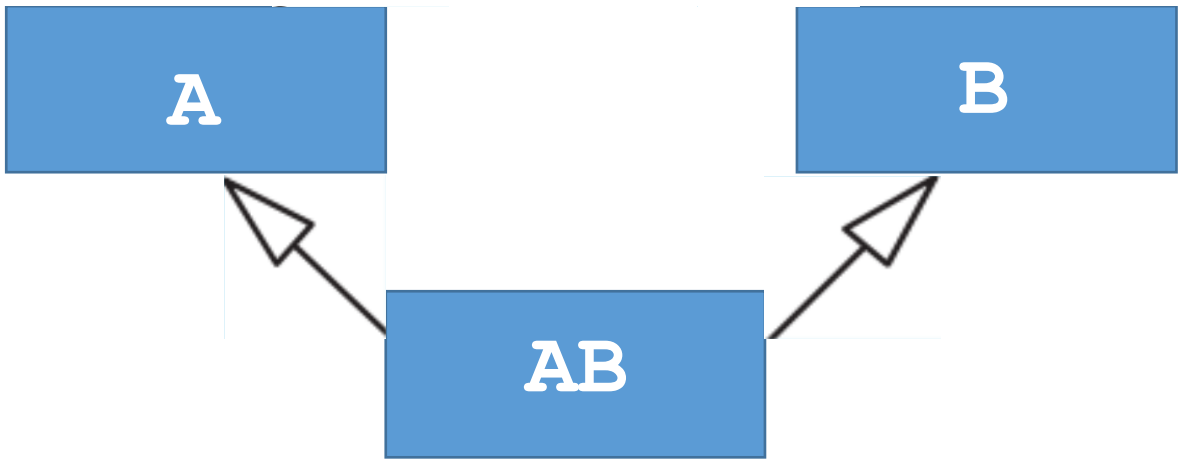
Multiple Inheritance
Diamond Inheritance

C++ Standard Library use diamond inheritance to form class `base_iostream`.



Multiple Inheritance
Diamond Inheritance

If we add a base class to our example – a new base class above our existing `A` and `B` in the hierarchy…



```
class O
{
    public :
        O(string myname="")
        {
            name = myname;
        }
        string name;
};

class A : public O
{
    public :
        A(string myname="")
        {
            name = myname;
        }
        string name;
};
```

```
class B : public O
{
    public :
        B(string myname="")
        {
            name = myname;
        }
        string name;
};

class AB : public A, public B
{
    public :
        AB(string myname="")
        {
            A::name = "A"+myname;
            B::name = "B"+myname;
        }
        string ABVar;
};
```

```
(gdb) p a
$4 = {
  <O> = {
    name = ""
  },
  members of A:
  name = "IamA"
}
(gdb) p b
$5 = {
  <O> = {
    name = ""
  },
  members of B:
  name = "IamB"
}
```

```
(gdb) p ab
$6 = {
  <A> = {
    <O> = {
      name = ""
    },
    members of A:
    name = "AIamAB"
  },
  <B> = {
    <O> = {
      name = ""
    },
    members of B:
    name = "BIamAB"
  },
  members of AB:
  ABVar = ""
}
```

```
(gdb) p ab
$6 = {
  <A> = {
    <O> = {
      name = ""
    },
    members of A:
    name = "AIamAB"
  },
  <B> = {
    <O> = {
      name = ""
    },
    members of B:
    name = "BIamAB"
  },
  members of AB:
  ABVar = ""
}
```

Object `ab` has two copies of `O`

So which one will print?

```
cout << "Object A's name is " << a.name << endl;
cout << "Object B's name is " << b.name << endl;
cout << "Object AB's A name is " << ab.A::name << endl;
cout << "Object AB's B name is " << ab.B::name << endl;
cout << "Object AB's O name is " << ab.O::name << endl;
```

```
diamondDemo.cpp: In function 'int main()':
diamondDemo.cpp:60:44: error: 'O' is an ambiguous base of 'AB'
  cout << "Object AB's O name is " << ab.O::name << endl;
                                            ^
```

```
(gdb) p ab
$6 = {
  <A> = {
    <O> = {
      name = ""
    },
    members of A:
    name = "AIamAB"
  },
  <B> = {
    <O> = {
      name = ""
    },
    members of B:
    name = "BIamAB"
  },
  members of AB:
  ABVar = ""
}
```

```
cout << "Object AB's O name is " << ab.O::name << endl;
```

```
diamondDemo.cpp: In function 'int main()':
diamondDemo.cpp:60:44: error: 'O' is an ambiguous base of 'AB'
  cout << "Object AB's O name is " << ab.O::name << endl;
                                           ^
```

```
cout << "Object AB's O name is " << ab.A::O::name << endl;
```

```
student@cse1325:/media/sf_VM@ make
g++ -c -g -std=c++11  diamondDemo.cpp -o diamondDemo.o
diamondDemo.cpp: In function 'int main()':
diamondDemo.cpp:58:47: error: 'O' is an ambiguous base
of 'AB'
  cout << "Object AB's O name is " << ab.A::O::name <<
endl;
                                              ^~~~
makefile:14: recipe for target 'diamondDemo.o' failed
make: *** [diamondDemo.o] Error 1
```

Multiple Inheritance
Diamond Inheritance

How to resolve the ambiguity

```
class A :          public O
{
   public :
      A(string myname="")
      {
         name = myname;
      }
      string name;
};
```

```
class B :          public O
{
   public :
      B(string myname="")
      {
         name = myname;
      }
      string name;
};
```

```
(gdb) p a
$1 = {
  <O> = {
    name = ""
  },
  members of A:
  _vptr.A = 0x401fc0
<VTT for A>,
  name = "IamA"
}
(gdb) p b
$2 = {
  <O> = {
    name = ""
  },
  members of B:
  _vptr.B = 0x401fa0
<VTT for B>,
  name = "IamB"
}
```

```
(gdb) p ab
$3 = {
  <A> = {
    <O> = {
      name = ""
    },
    members of A:
    _vptr.A = 0x401f20 <vtable for AB+24>,
    name = "AIamAB"
  },
  <B> = {
    members of B:
    _vptr.B = 0x401f38 <VTT for AB>,
    name = "BIamAB"
  },
  members of AB:
  ABVar = ""
}
```

Only one copy of O in object ab now.

```
Object A's name is IamA
Object B's name is IamB
Object AB's A name is AIamAB
Object AB's B name is BIamAB
Object AB's O name is
```