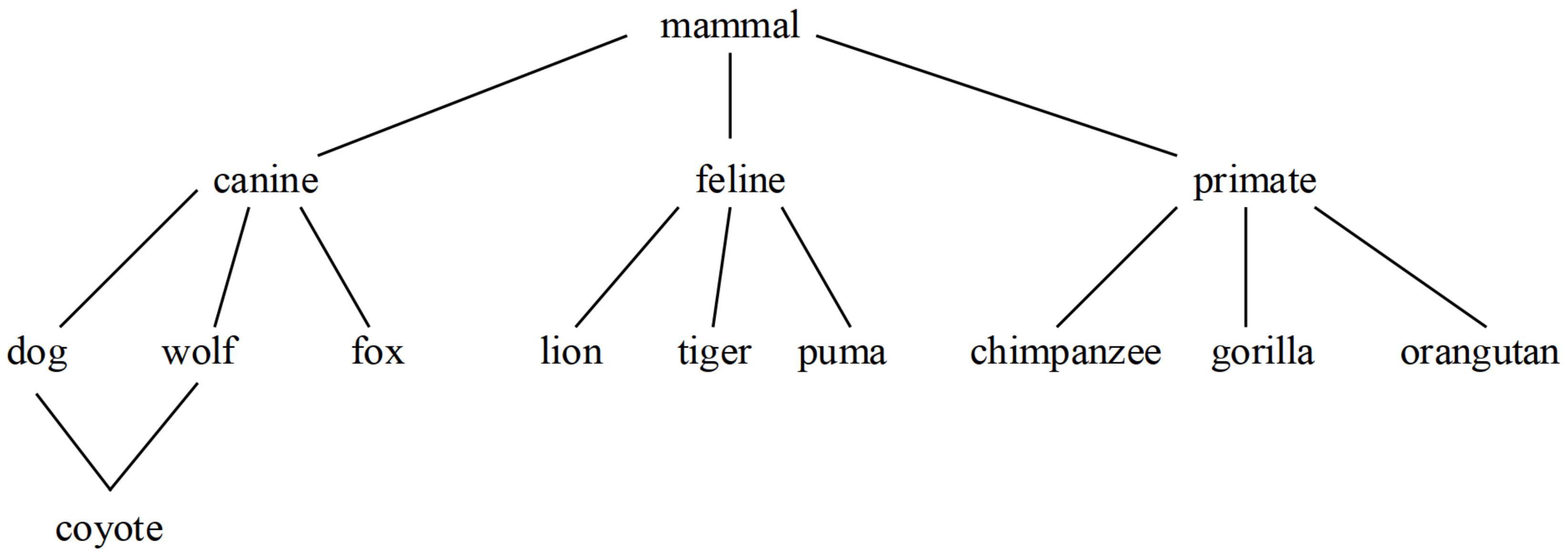
Inheritance

- Inheritance is a relationship among classes where a subclass inherits the structure and behavior of its super-class.
  - Defines the “is a” or generalization/specialization hierarchy.
  - Structure: instance variables.
  - Behavior: instance methods.

Inheritance in C++

- C++ supports single and multiple inheritance



Class Derivation (Inheritance)

- In order to derive a class, the following two extensions to the class syntax are necessary
  - class heading is modified to allow a derivation list of classes from which to inherit members.
  - An additional class level, that of ***protected***, is provided. A protected class member behaves as a public member to a derived class

```
class Cat : public Animal
{
   protected:

   // data members

};
```

Deriving a Class

class DerivedClass : access_specifier BaseClass

Constructor and Destructor Behavior

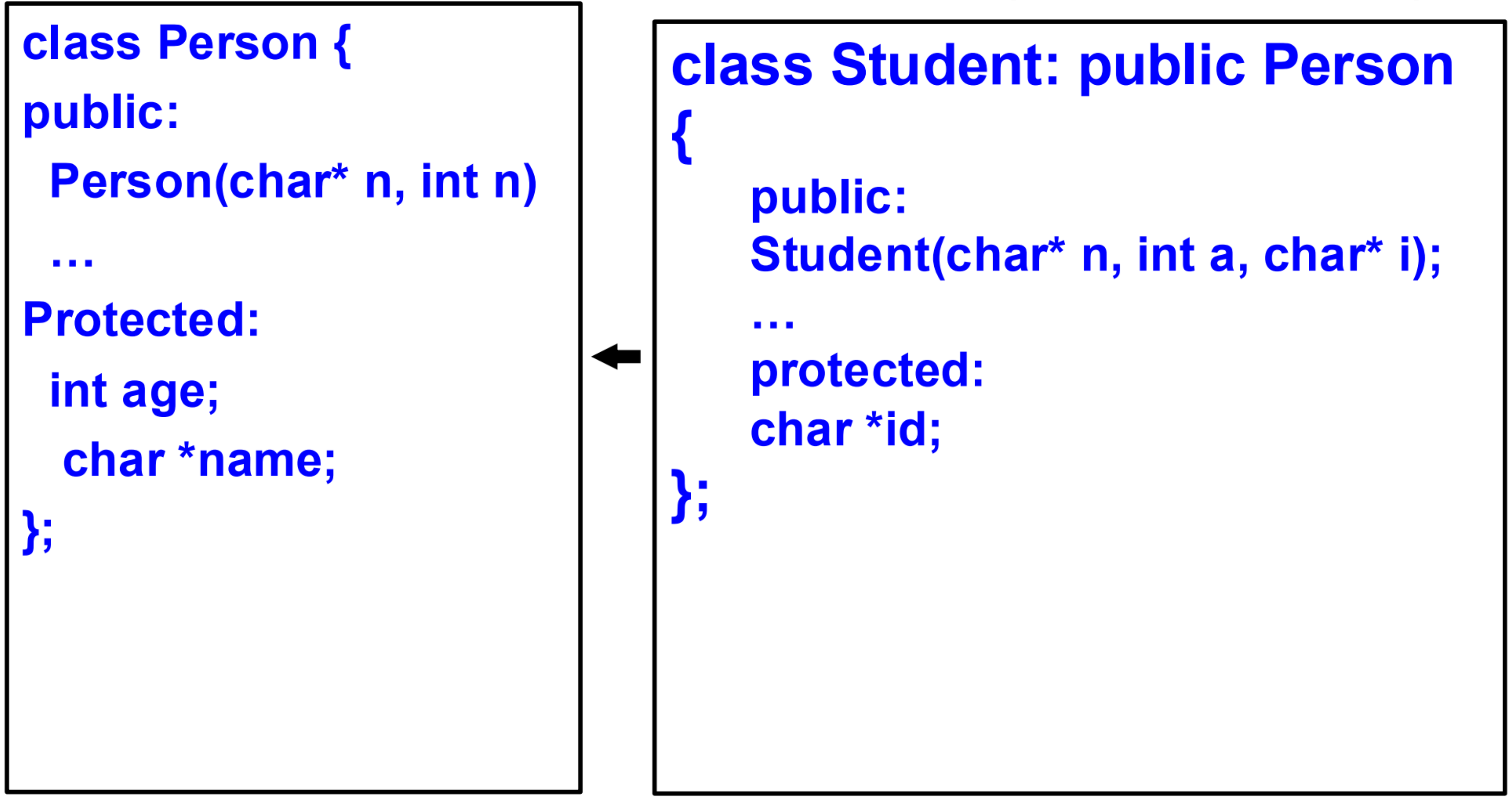
```
class Base {
public:
   Base() {
      cout << "Base class constructor called" << endl;
   }
   ~Base() {
      cout << "Base class destructor called" << endl;
   }
};
```

```
class Derived : public Base {
public:
   Derived() {
      cout << "Derived class constructor called" << endl;
   }
   ~Derived() {
      cout << "Derived class destructor called" << endl;
   }
};
```

```
Base class constructor called
Derived class constructor called
Derived class destructor called
Base class destructor called
```

```
int main() {
   Derived d; // Creating an object of Derived class
   return 0;
}
```

Class derivation - Example



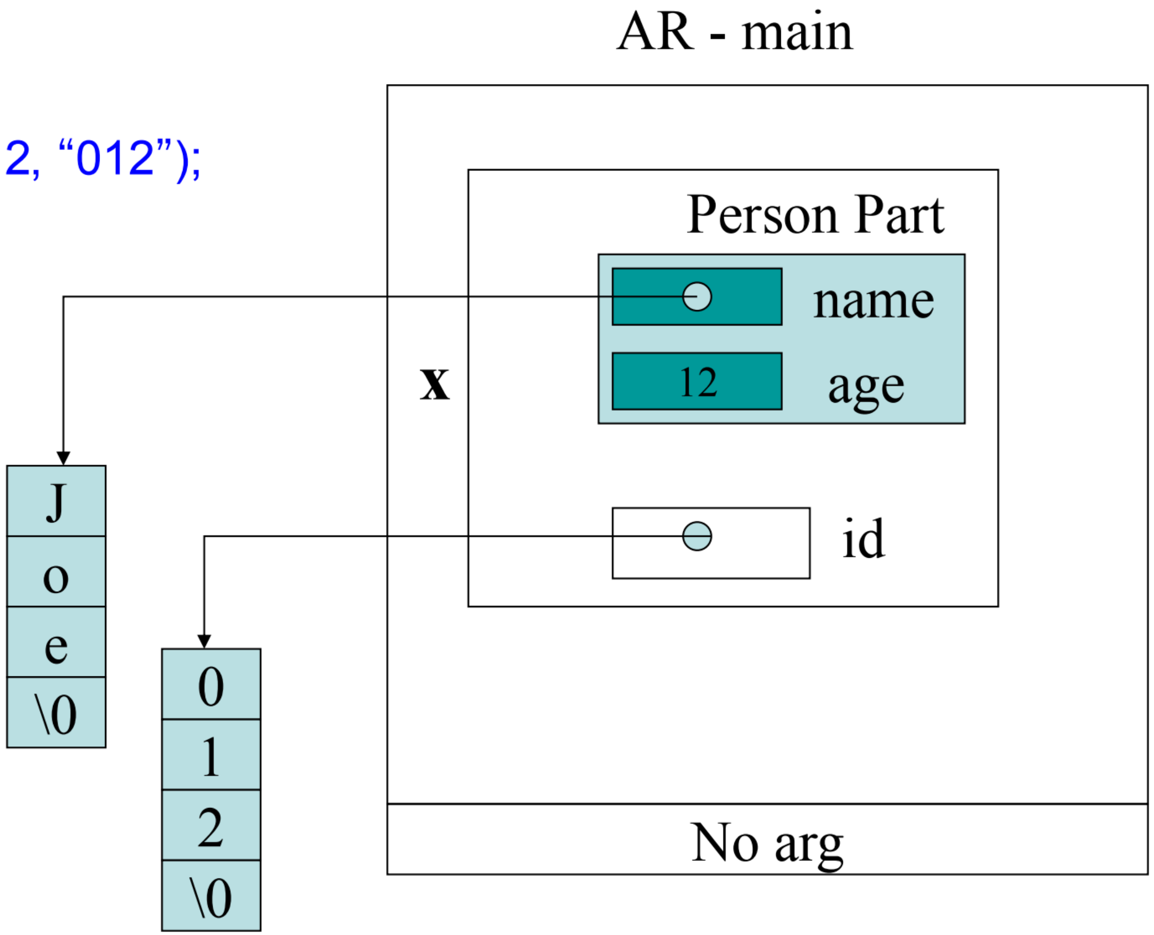
Student(char* n, int a, char* i) : Person(n, a)

```
class Person {
public:
  Person(char* n, int n)
  …
Protected:
  int age;
   char *name;
};
```

```
class Student: public Person
{
      public:
      Student(char* n, int a, char* i);
      …
      protected:
      char *id;
};
```

Example Continued

```
int main ()
{
   Student  x ("Joe", 12, "012");
    return 0;
}
```



Base Class Design

- Syntax for defining a base class is the same as an ordinary class with two exceptions:
  - Members intended to be inherited but not intended to be public are declared as ***protected*** members.
- Member functions whose implementation depends on representational details of subsequent derivations that are unknown at the time of the base class design are declared as *virtual functions*.

Protected

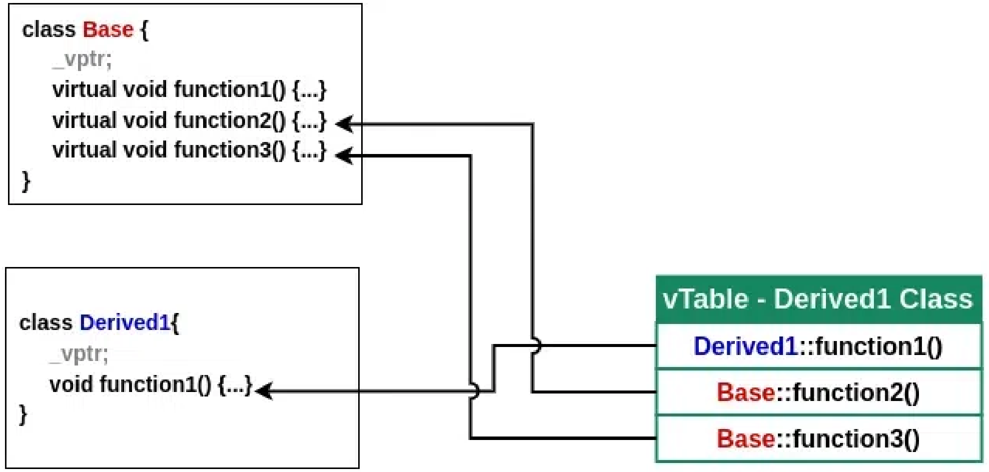
```
class Base {
protected:
    int protectedData;
};

class Derived : public Base {
public:
    void accessBaseData() {
        // Can access protectedData because it's inherited as protected
        protectedData = 10;
    }
};
```

Virtual Functions

```
class Base {
public:
   virtual void display() {
      cout << "Display from Base" << endl;
   }
};

class Derived : public Base {
public:
   void display() override {
      cout << "Display from Derived" << endl;
   }
};
```



```
class Base {
    _vptr;
    virtual void function1() {...}
    virtual void function2() {...}
    virtual void function3() {...}
}
```

```
class Derived1{
    _vptr;
    void function1() {...}
}
```

vTable - Derived1 Class
Derived1::function1()
Base::function2()
Base::function3()

```
int main() {
   Base* basePtr = new Derived();
   basePtr->display(); // Calls Derived's display() due to
virtual function
   delete basePtr;
}
```

Why Virtual Functions are Useful

- **Dynamic Behavior**
- **Base Class Design Strategy**

Base Class Design (Continued)

```
class Person {
   public:
      Person();
      virtual ~Person();
      virtual display();
      …
   protected:
      int age;
      char *name;
};
```

Person* person = new Student(); // Student is a class derived from Person
delete person; // If ~Person() is not virtual, Student's destructor won't be called!

Inherited member access

- The derived class member functions can have access to inherited members directly or by using the the scope resolution operator:

```
void Student :: display() {
    cout << Person::name << age;
    }
```

```
void Student::display() {
   cout << name << " " << age;  // Direct access to name
and age
}
```

In this example name also could be accessed directly without using scope resolution operator.

Inherited Member Access (Continued)

- In most cases, use of the class scope resolution operator is redundant. In two cases, however, using scope resolution operator is necessary:
  1. When an inherited member’s name is reused in the derived class.
  2. When two or more base classes define an inherited member with the same name.

Case 1: When an Inherited Member’s Name is Reused in the Derived Class:

```
class Person {
protected:
   char* name;
};

class Student : public Person {
protected:
   char* name;  // This hides Person::name
public:
   void showName() {
      cout << Person::name;  // Use scope resolution to access Person::name
      cout << name;          // Access the Student::name
   }
};
```

Case 2: When Two or More Base Classes Define an Inherited Member with the Same Name

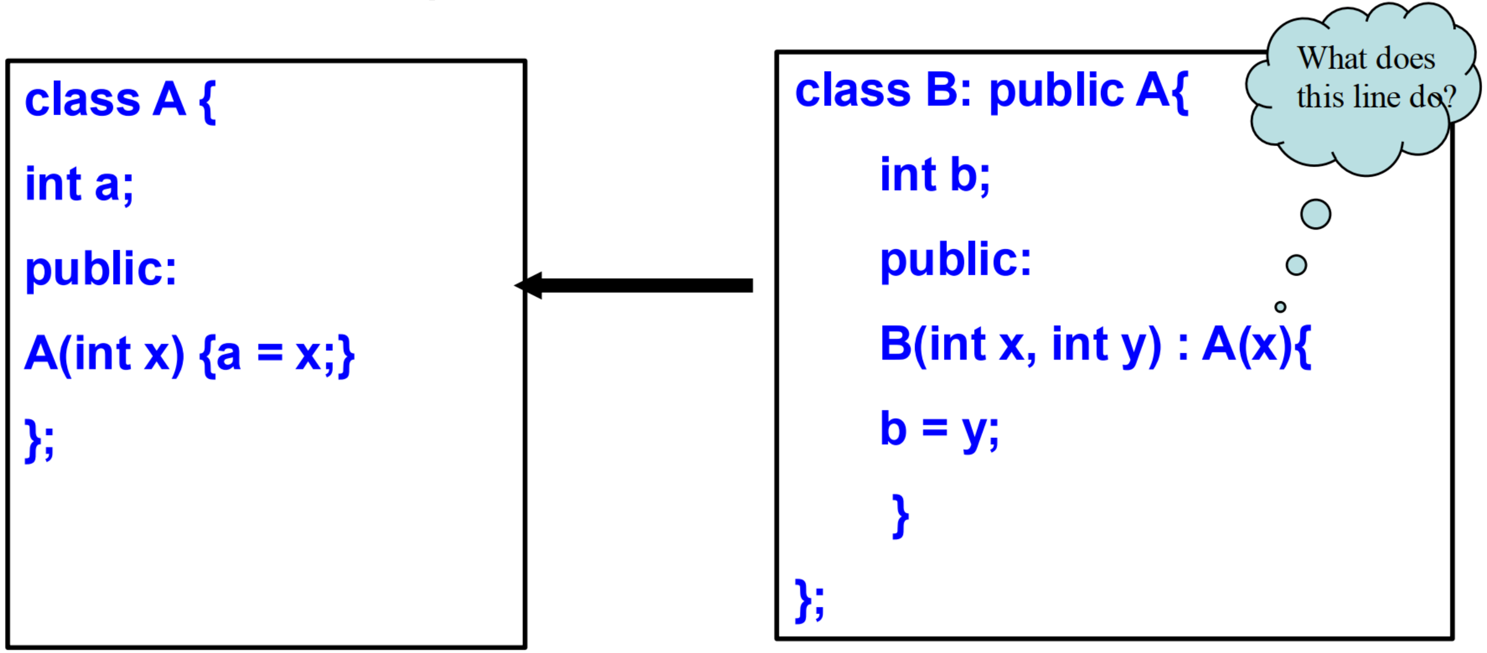
```
class Teacher {
protected:
  int id;
};
```

```
class Admin {
protected:
  int id;
};
```

```
class Principal : public Teacher, public Admin {
public:
  void showID() {
    cout << Teacher::id;  // Access Teacher's id
    cout << Admin::id;    // Access Admin's id
  }
};
```

Base Class Initialization

- Member initialization list is used to pass arguments to a base class constructor. The tag name of a base class is specified, followed by its argument list enclosed in parentheses.



```
class A {
int a;
public:
A(int x) {a = x;}
};
```

```
class B: public A{
      int b;
      public:
      B(int x, int y) : A(x){
      b = y;
       }
};
```

What does this line do?

Special Relationship between Base and Derived Class

- A derived class can be assigned to any of its public base classes without requiring an explicit cast.

  For example, consider class Student is derived from class Person and class Monitor is derived from class Student :

```
Person x;
Student y;
Monitor z;
x = y;                     // OK
y =  (Student) x;                   // Needs cast
x = z;                     // OK
```

- A derived class can be assigned to any of its public base classes without requiring an explicit cast. How is this feature related to polymorphism?