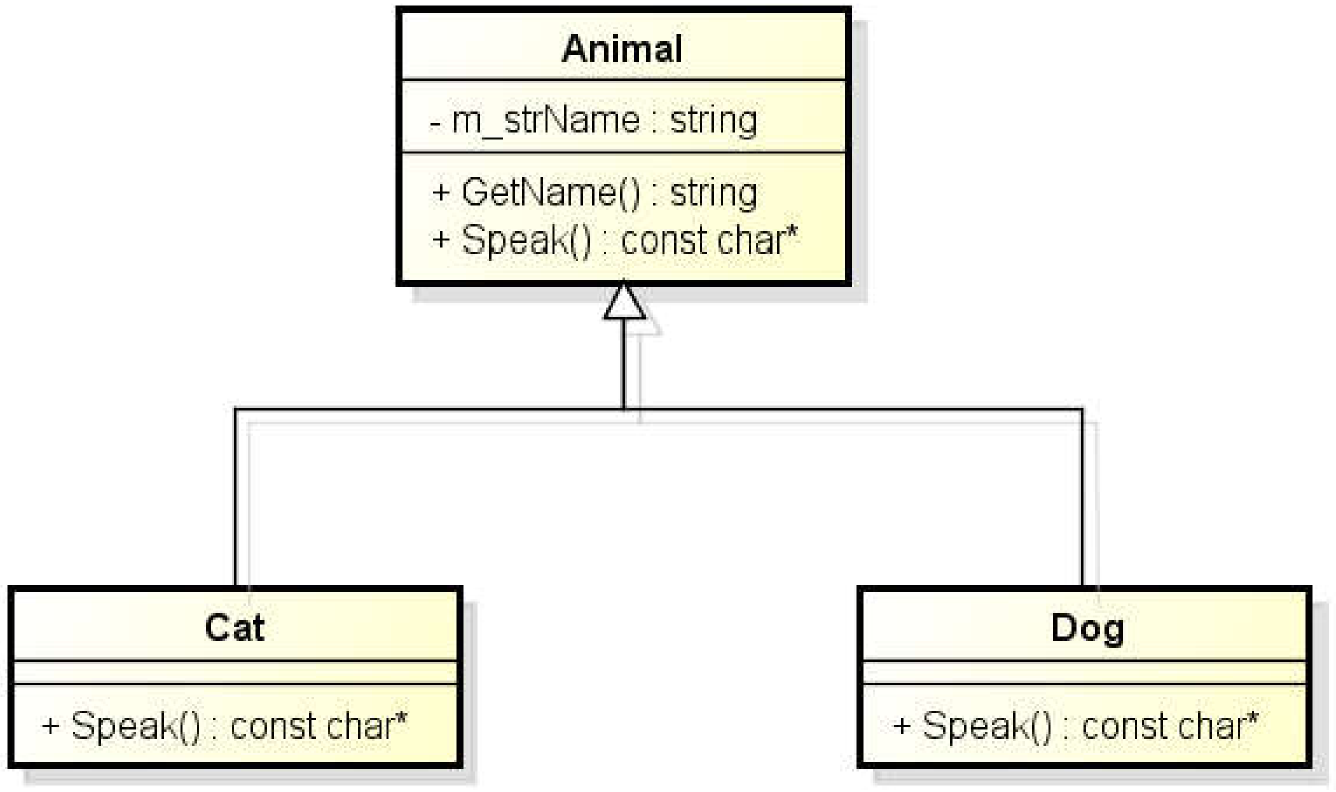
Introduction to Polymorphism

ThanhNT

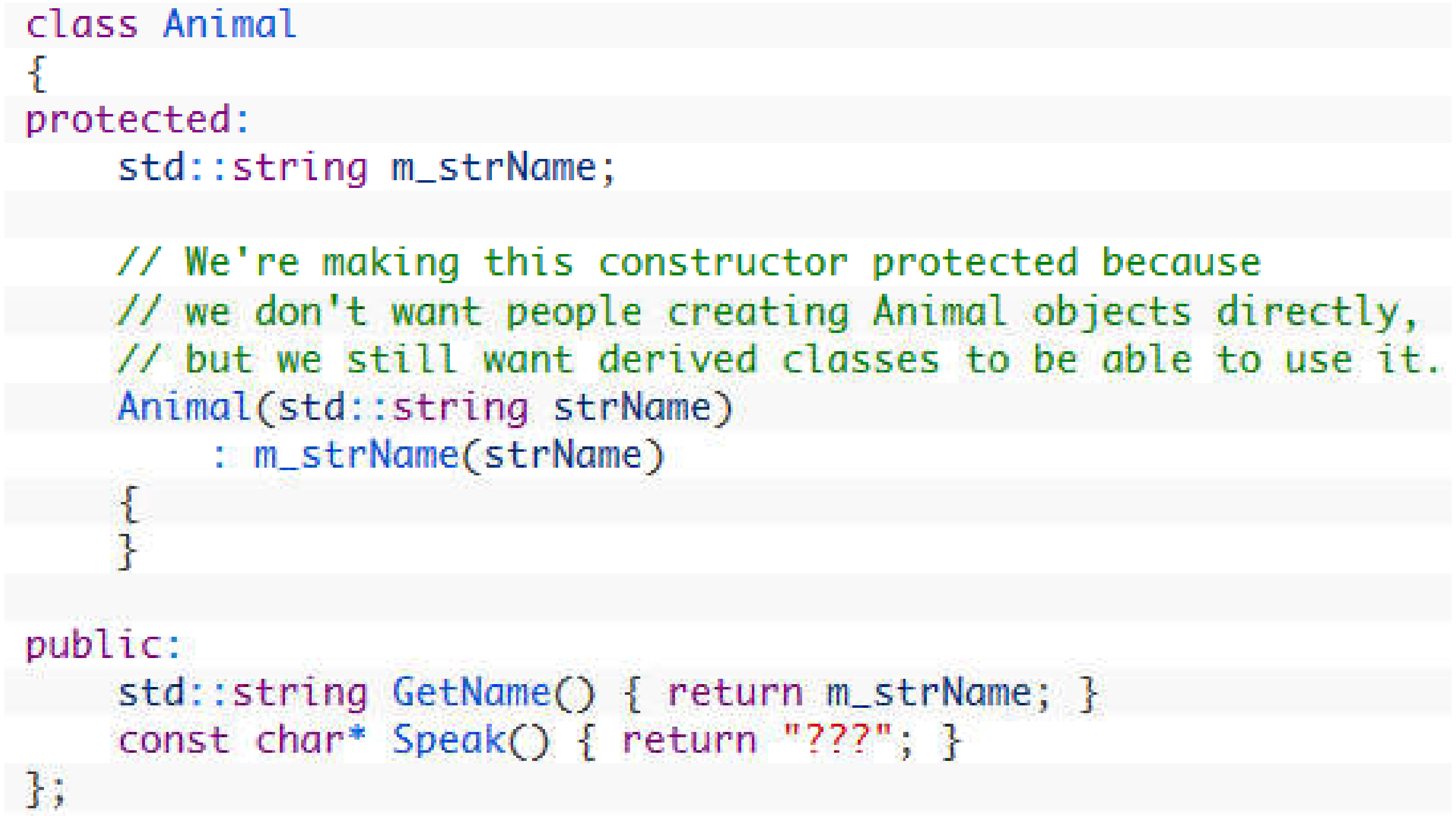
Introduction to Polymorphism

- Consider an example of derived class.



Introduction to Polymorphism

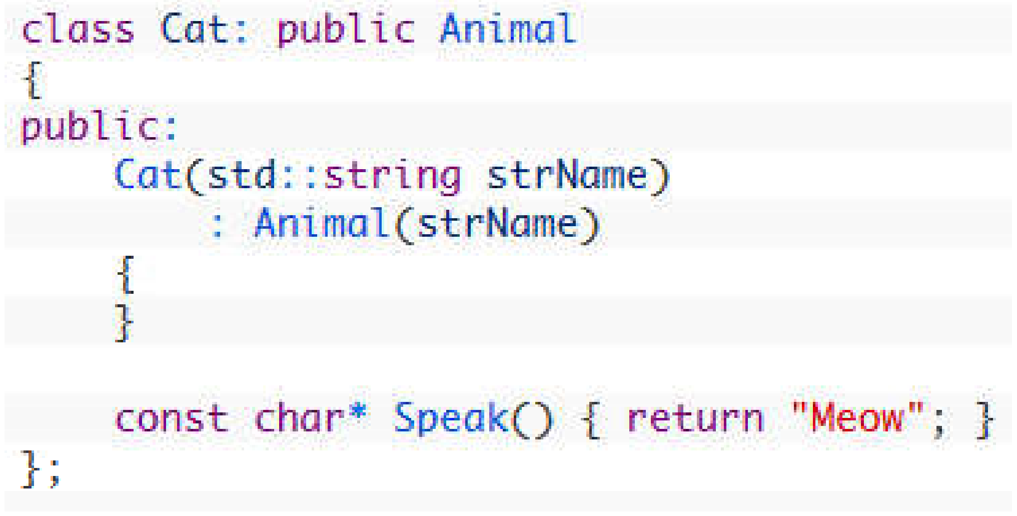
- Animal class definition.



```
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    const char* Speak() { return "???"; }
};
```

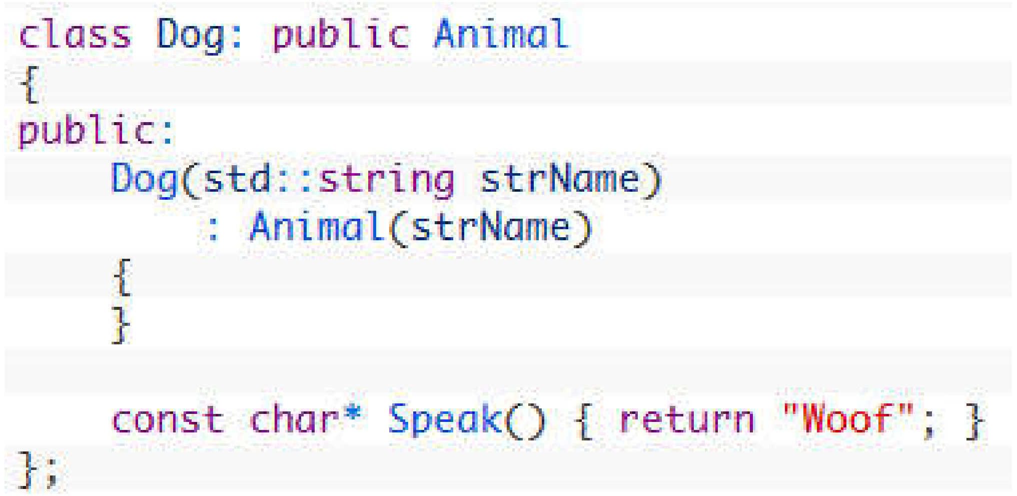
Introduction to Polymorphism

- Cat class definition:



```
class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    const char* Speak() { return "Meow"; }
};
```

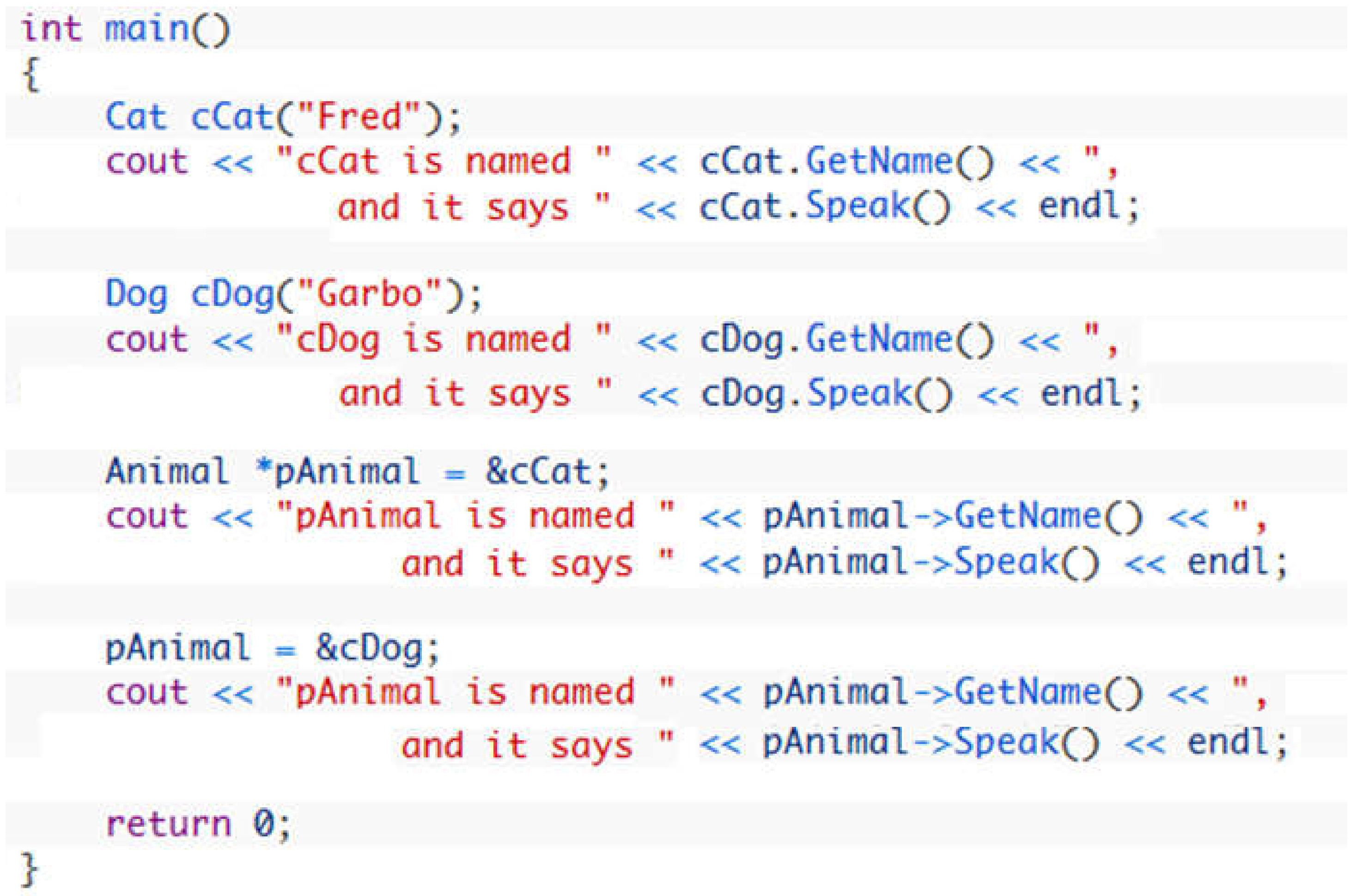
- Dog class definition:



```
class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    const char* Speak() { return "Woof"; }
};
```

Introduction to Polymorphism

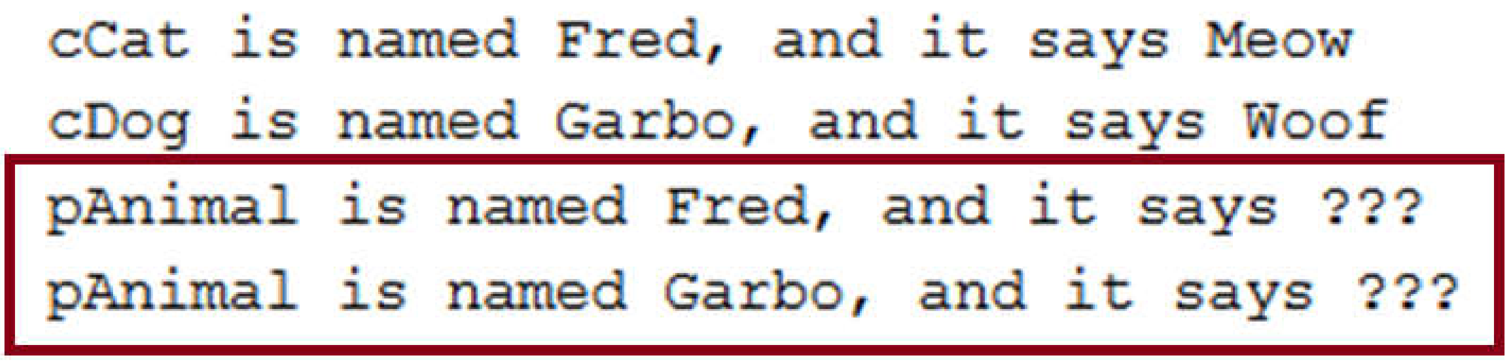
- Main function of using these classes.



```
int main()
{
    Cat cCat("Fred");
    cout << "cCat is named " << cCat.GetName() << ",
                 and it says " << cCat.Speak() << endl;

    Dog cDog("Garbo");
    cout << "cDog is named " << cDog.GetName() << ",
                 and it says " << cDog.Speak() << endl;

    Animal *pAnimal = &cCat;
    cout << "pAnimal is named " << pAnimal->GetName() << ",
                   and it says " << pAnimal->Speak() << endl;

    pAnimal = &cDog;
    cout << "pAnimal is named " << pAnimal->GetName() << ",
                   and it says " << pAnimal->Speak() << endl;

    return 0;
}
```

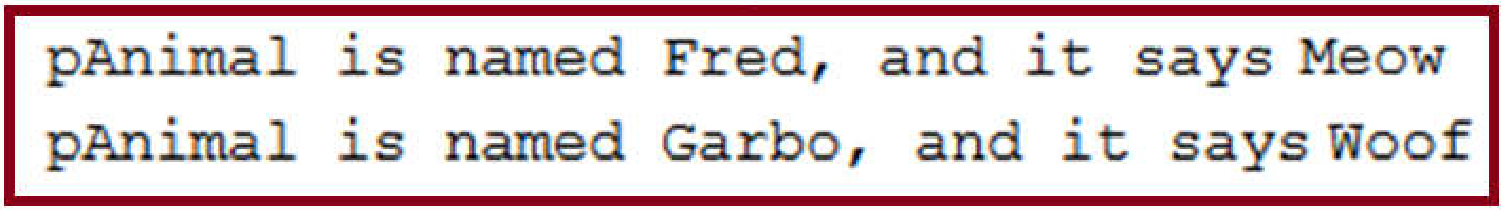
Introduction to Polymorphism

- Result:



```
cCat is named Fred, and it says Meow
cDog is named Garbo, and it says Woof
pAnimal is named Fred, and it says ???
pAnimal is named Garbo, and it says ???
```

- Expected:

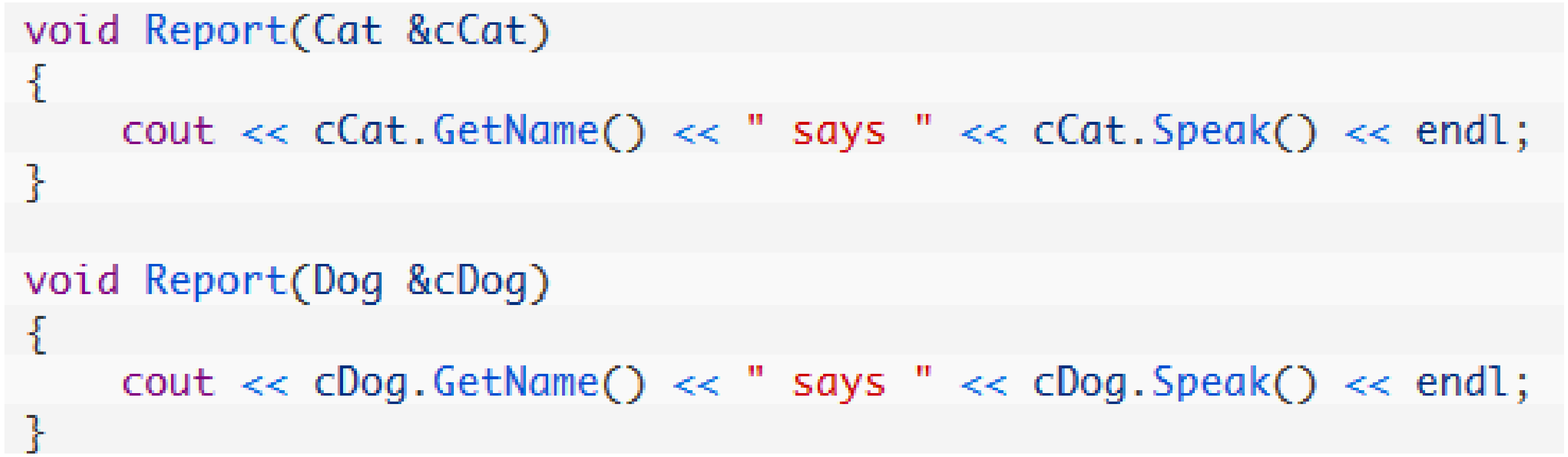


```
pAnimal is named Fred, and it says Meow
pAnimal is named Garbo, and it says Woof
```

- Issue: Because pAnimal is an Animal pointer, it can only see the Animal class. Consequently, pAnimal->Speak() calls Animal::Speak() rather than the Dog::Speak() or Cat::Speak() function.

Introduction to Polymorphism

- Solution:

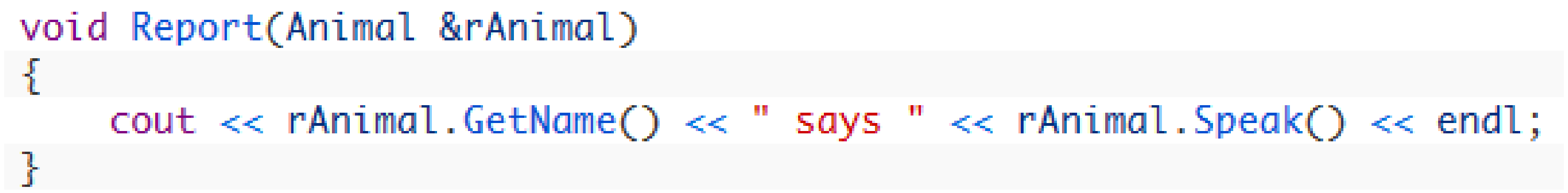


```cpp
void Report(Cat &cCat)
{
    cout << cCat.GetName() << " says " << cCat.Speak() << endl;
}

void Report(Dog &cDog)
{
    cout << cDog.GetName() << " says " << cDog.Speak() << endl;
}
```

- Disadvantage: If we have 30 different animal types instead of 2, you'll have to write 30 almost identical functions.

Introduction to Polymorphism

- Because Cat and Dog are derived from Animal, it makes sense that we should be able to do something like this:

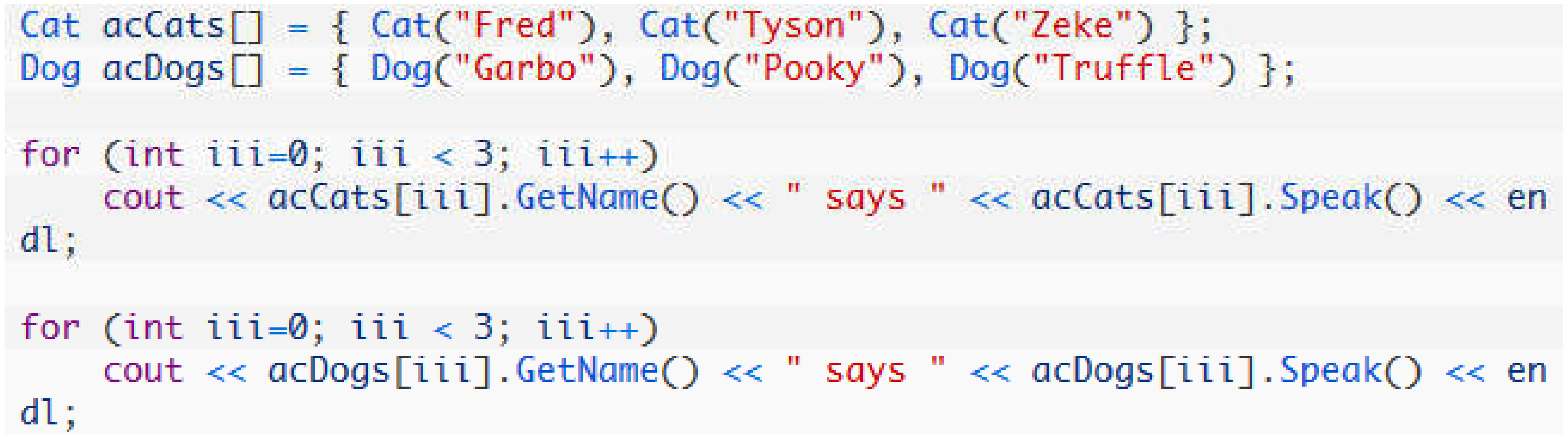


```
void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}
```

- How we can do that while we have the issue of calling parent method instead of child method?

Introduction to Polymorphism

- If we have many cats and dogs:

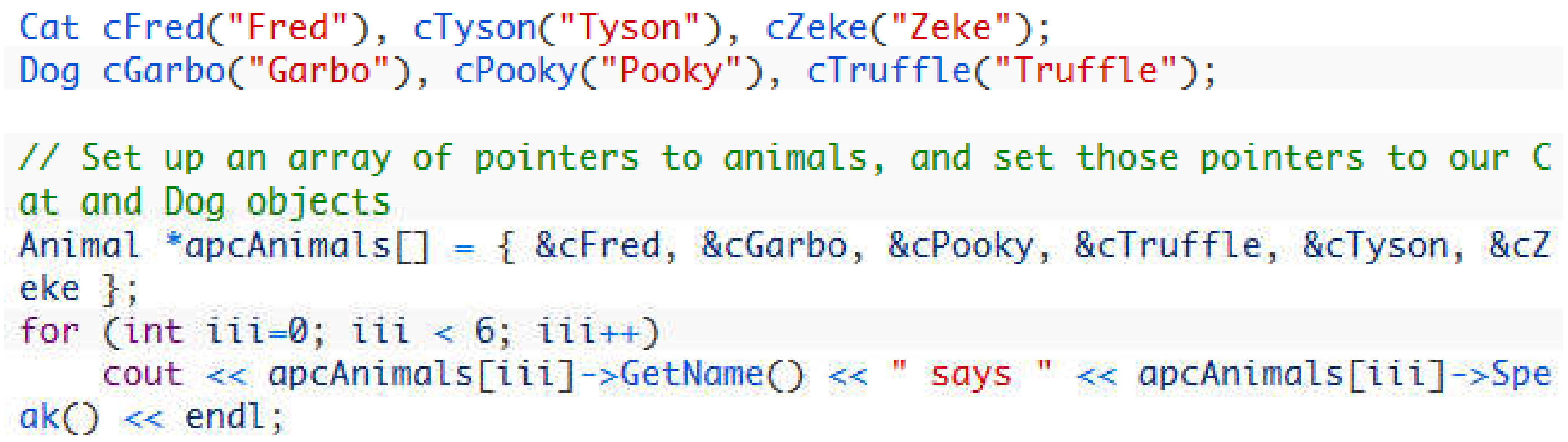


```
Cat acCats[] = { Cat("Fred"), Cat("Tyson"), Cat("Zeke") };
Dog acDogs[] = { Dog("Garbo"), Dog("Pooky"), Dog("Truffle") };

for (int iii=0; iii < 3; iii++)
    cout << acCats[iii].GetName() << " says " << acCats[iii].Speak() << en
dl;

for (int iii=0; iii < 3; iii++)
    cout << acDogs[iii].GetName() << " says " << acDogs[iii].Speak() << en
dl;
```

- Disadvantage: If we have 30 different animal types instead of 2, you'll need 30 arrays.

Introduction to Polymorphism

- Because both Cat and Dog are Animal, it makes sense that we should be able to do something like this:



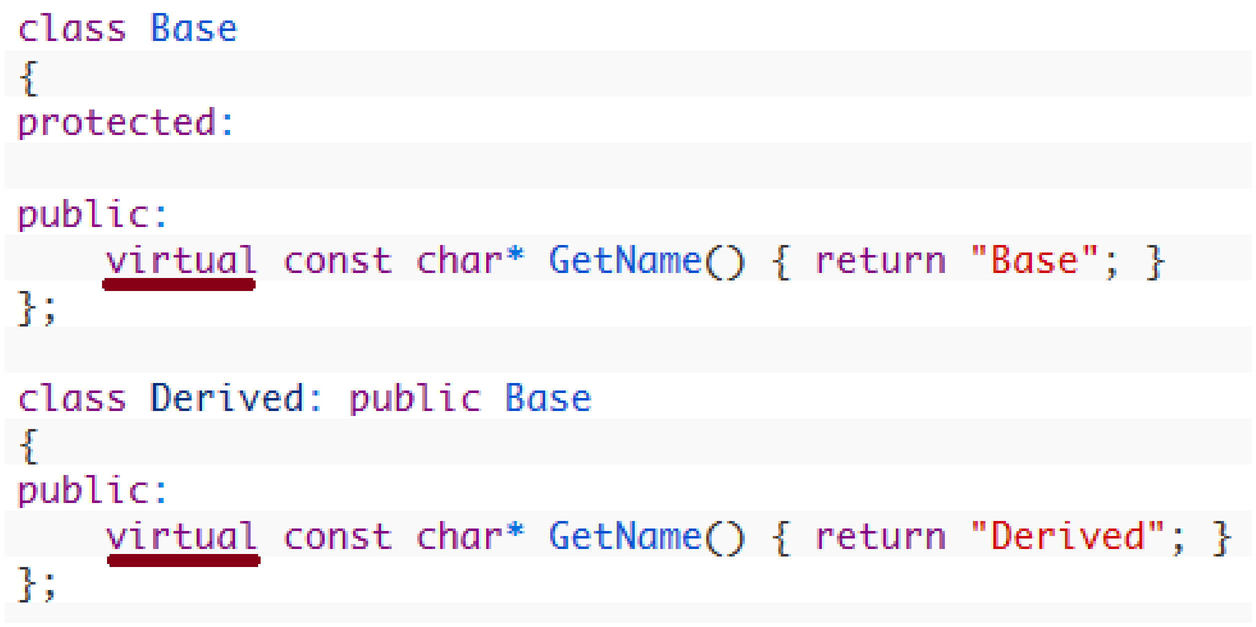
```
Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");

// Set up an array of pointers to animals, and set those pointers to our C
at and Dog objects
Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZ
eke };
for (int iii=0; iii < 6; iii++)
    cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Spe
ak() << endl;
```

- How we can do that while we have the issue of calling parent method instead of child method?

Virtual function

- A **virtual function** is a special type of function that resolves to the most-derived version of the function with the same signature. This capability is known as **polymorphism**.

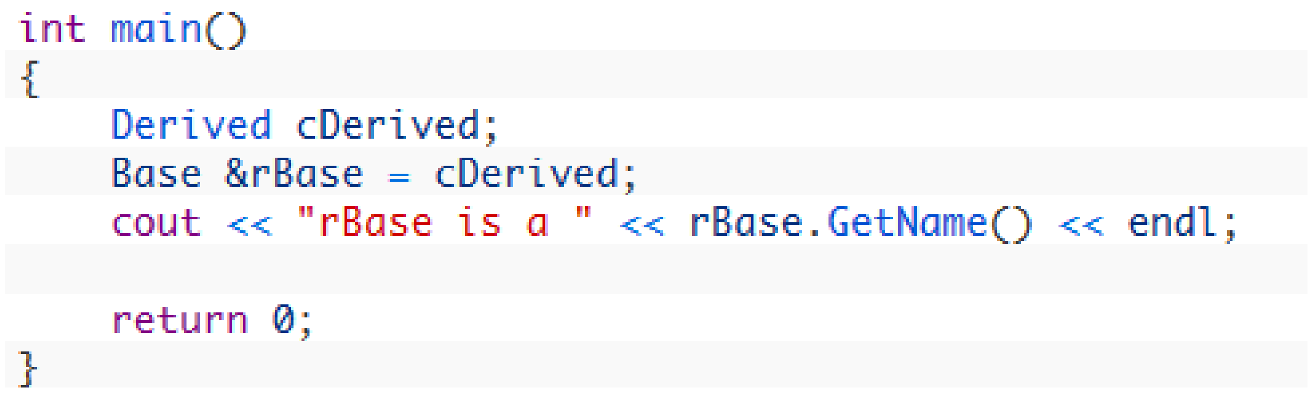
- To make a function virtual, simply place the "**virtual**" keyword before the function declaration.



```
class Base
{
protected:

public:
    virtual const char* GetName() { return "Base"; }
};

class Derived: public Base
{
public:
    virtual const char* GetName() { return "Derived"; }
};
```

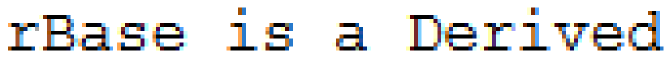
Virtual function

- Usage:



```
int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

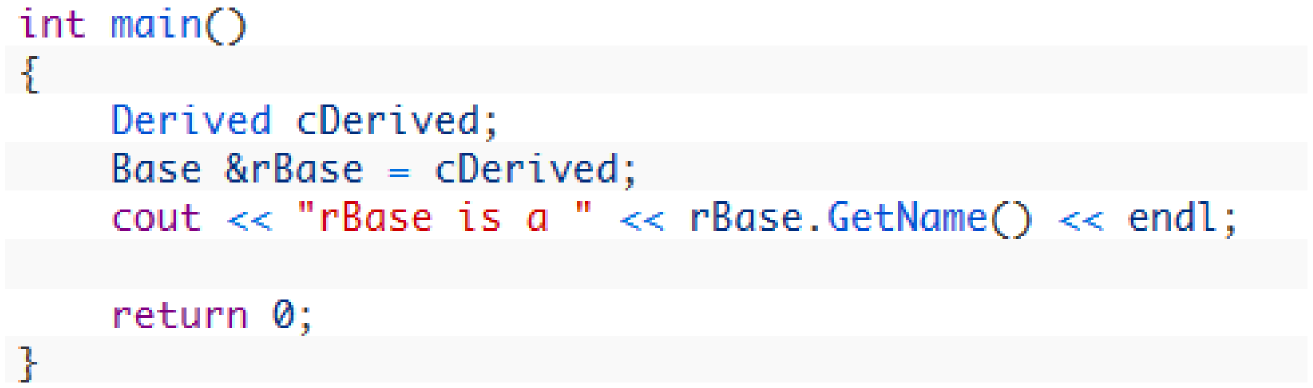
- Result:



```
rBase is a Derived
```

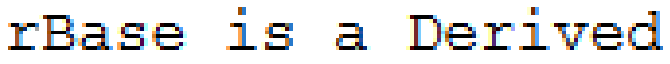
Virtual function

- Usage:



```
int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

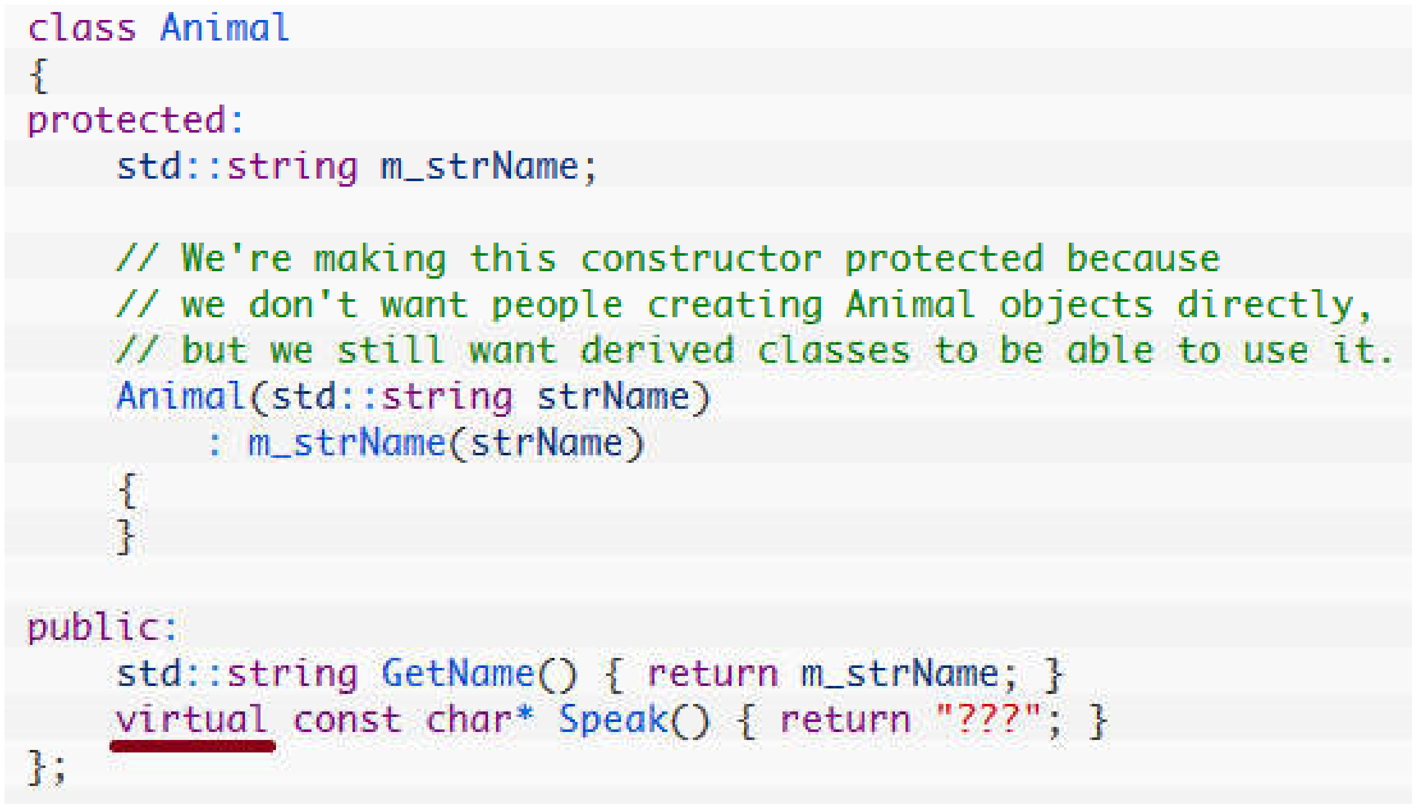
- Result:



```
rBase is a Derived
```

Introduction to Polymorphism

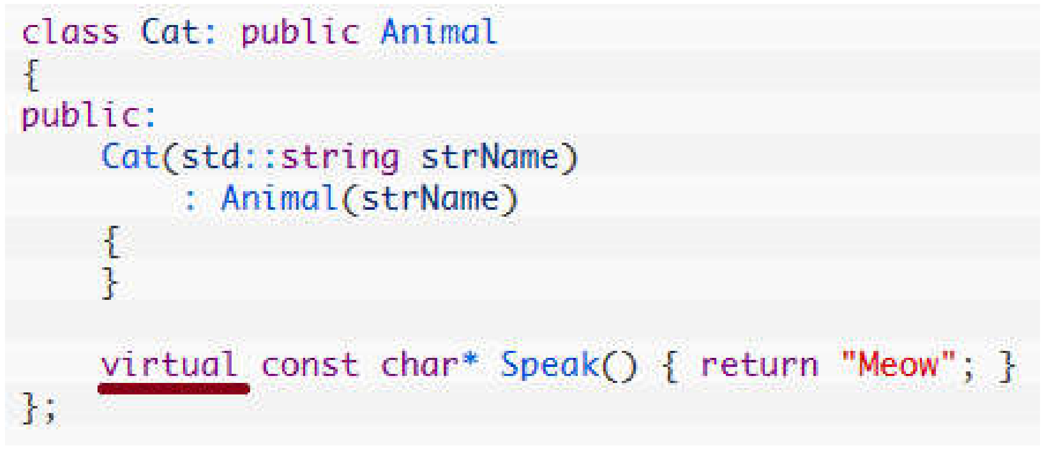
- Animal class definition with virtual function:



```cpp
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    virtual const char* Speak() { return "???"; }
};
```

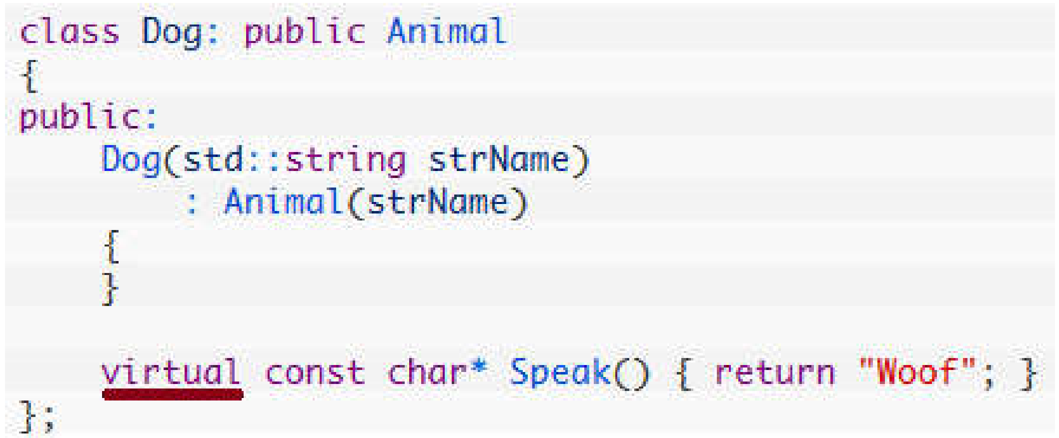
Introduction to Polymorphism

- Cat class definition with virtual function:



```
class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Meow"; }
};
```

- Dog class definition with virtual function:



```
class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Woof"; }
};
```

Introduction to Polymorphism

- Using these classes.

```
void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}

int main()
{
    Cat cCat("Fred");
    Dog cDog("Garbo");

    Report(cCat);
    Report(cDog);
}
```

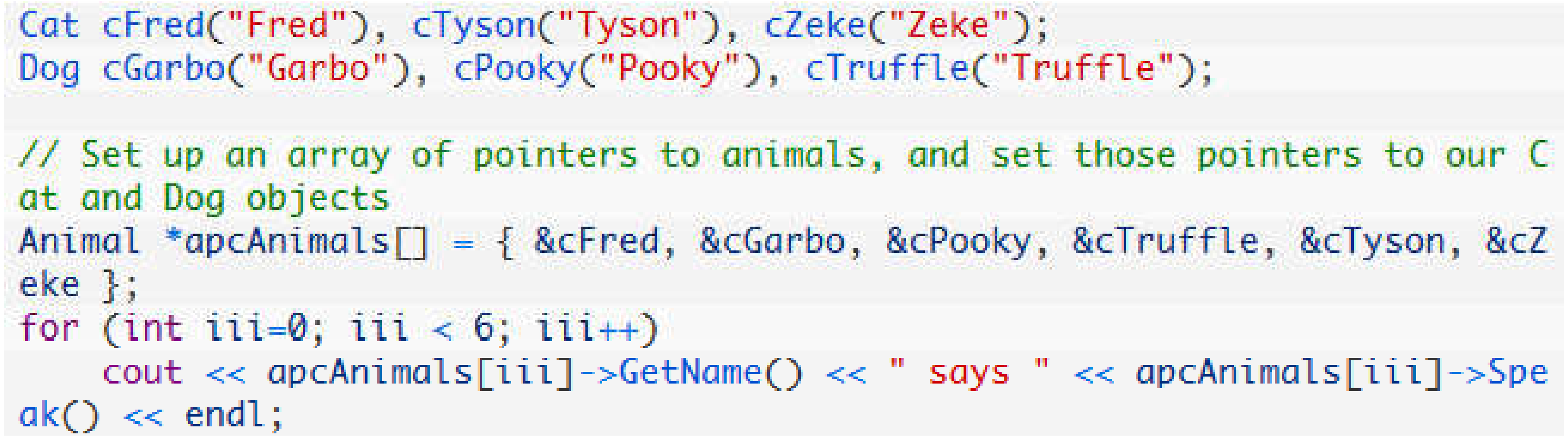
- Result:

```
Fred says Meow
Garbo says Woof
```

- Advantage: If we have 30 different animal types instead of 2, you'll need only one Report function.

Introduction to Polymorphism

- If we have many cats and dogs:



```
Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");

// Set up an array of pointers to animals, and set those pointers to our C
at and Dog objects
Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZ
eke };
for (int iii=0; iii < 6; iii++)
    cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Spe
ak() << endl;
```

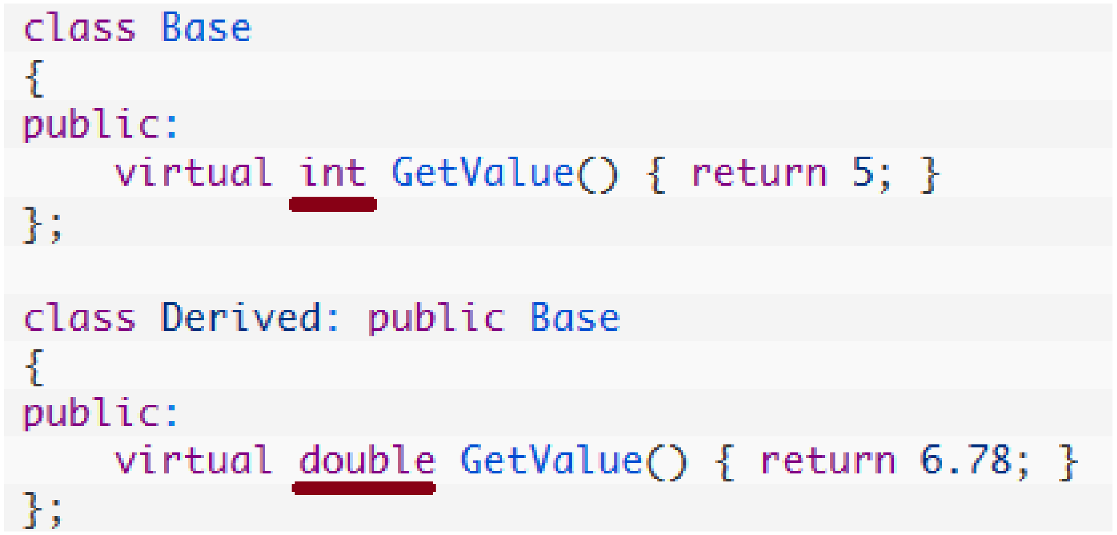
- Result:

```
Fred says Meow
Garbo says Woof
Pooky says Woof
Truffle says Woof
Tyson says Meow
Zeke says Meow
```

- Advantage: If we have 30 different animal types instead of 2, you’ll need only 1 array and 1 “for” loop.

Return types of virtual functions

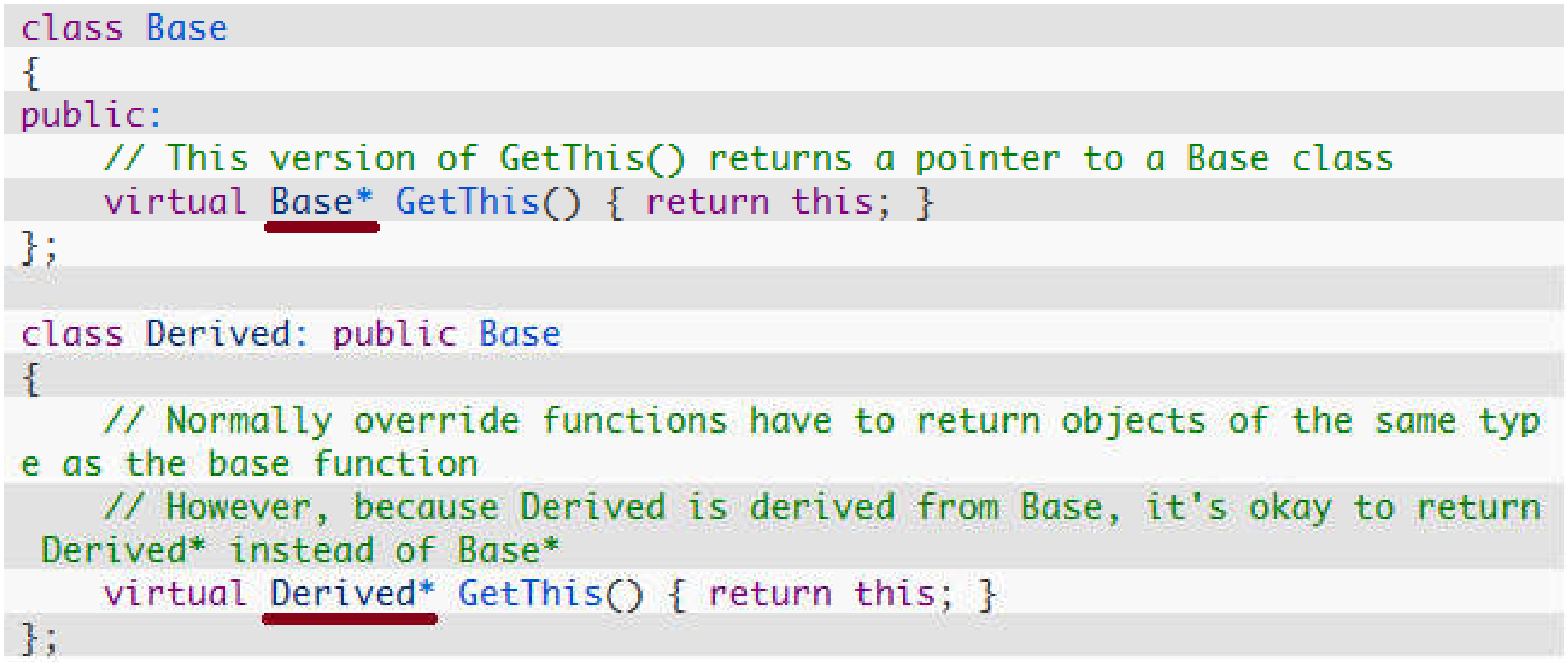
- The return type of a virtual function and its override must match. Thus the following will not work:



```
class Base
{
public:
    virtual int GetValue() { return 5; }
};

class Derived: public Base
{
public:
    virtual double GetValue() { return 6.78; }
};
```

Return types of virtual functions

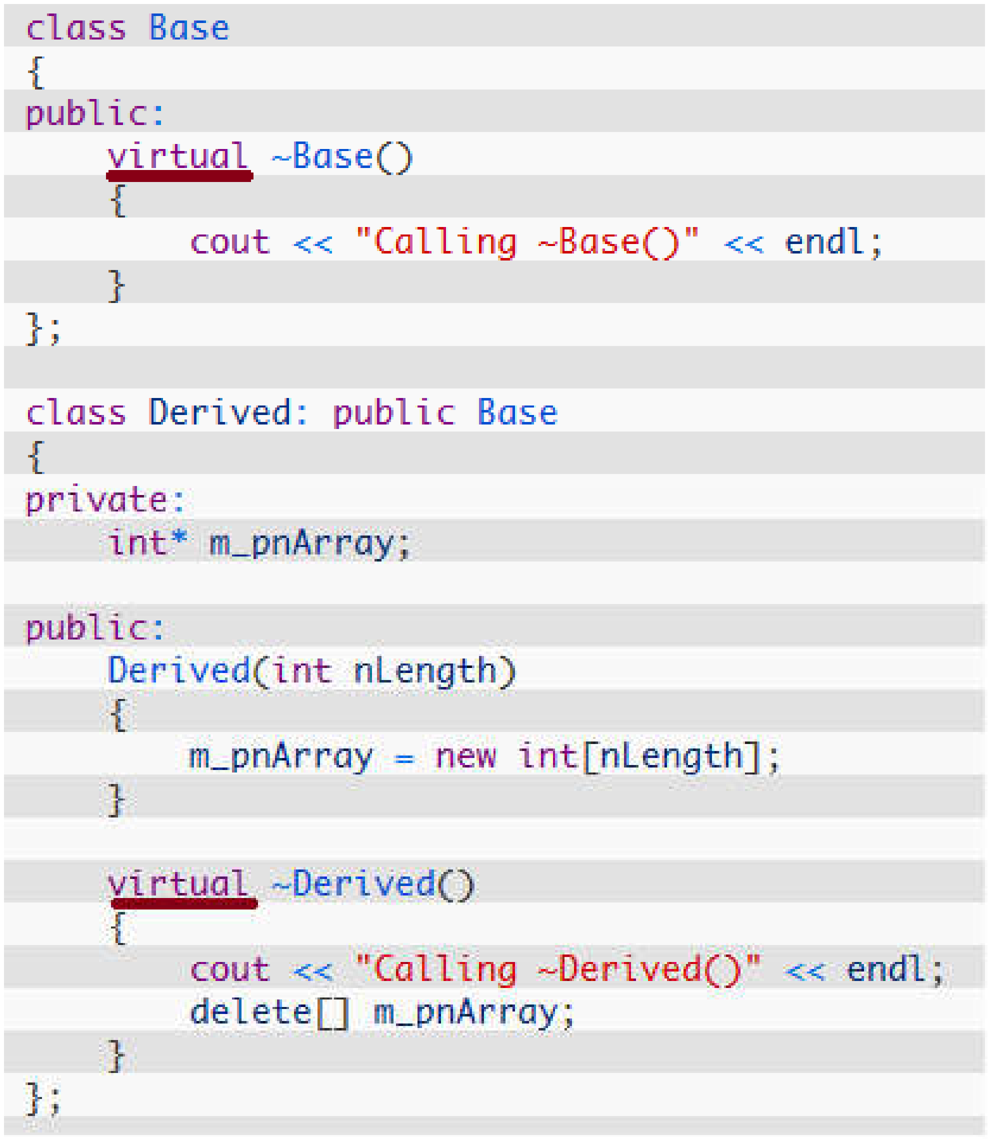
- Except a special case of pointer or a reference to a class:



```
class Base
{
public:
    // This version of GetThis() returns a pointer to a Base class
    virtual Base* GetThis() { return this; }
};

class Derived: public Base
{
    // Normally override functions have to return objects of the same typ
e as the base function
    // However, because Derived is derived from Base, it's okay to return
 Derived* instead of Base*
    virtual Derived* GetThis() { return this; }
};
```

Virtual destructors

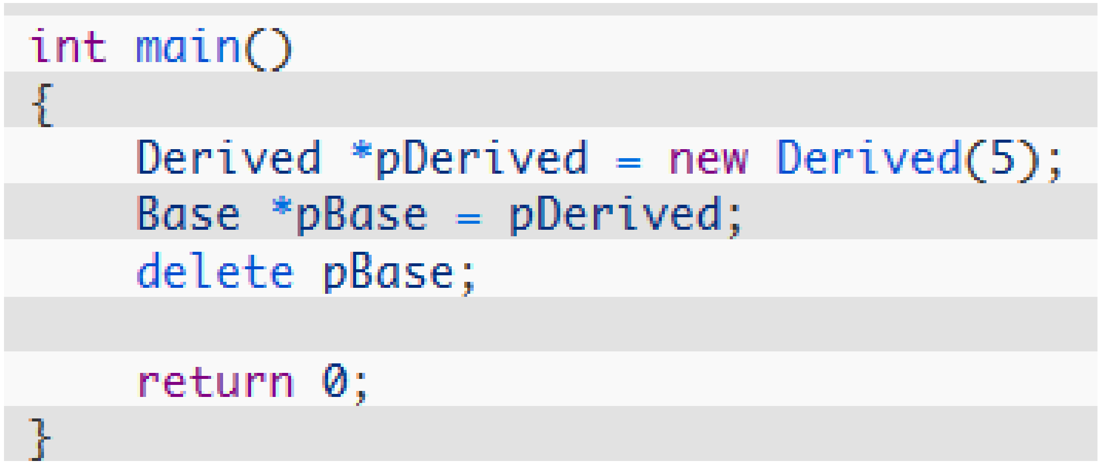
- Make Base's destructor virtual:



```
class Base
{
public:
    virtual ~Base()
    {
        cout << "Calling ~Base()" << endl;
    }
};

class Derived: public Base
{
private:
    int* m_pnArray;

public:
    Derived(int nLength)
    {
        m_pnArray = new int[nLength];
    }

    virtual ~Derived()
    {
        cout << "Calling ~Derived()" << endl;
        delete[] m_pnArray;
    }
};
```

Virtual destructors

- Delete function to call Derived’s destructor (which will call Base’s destructor in turn):



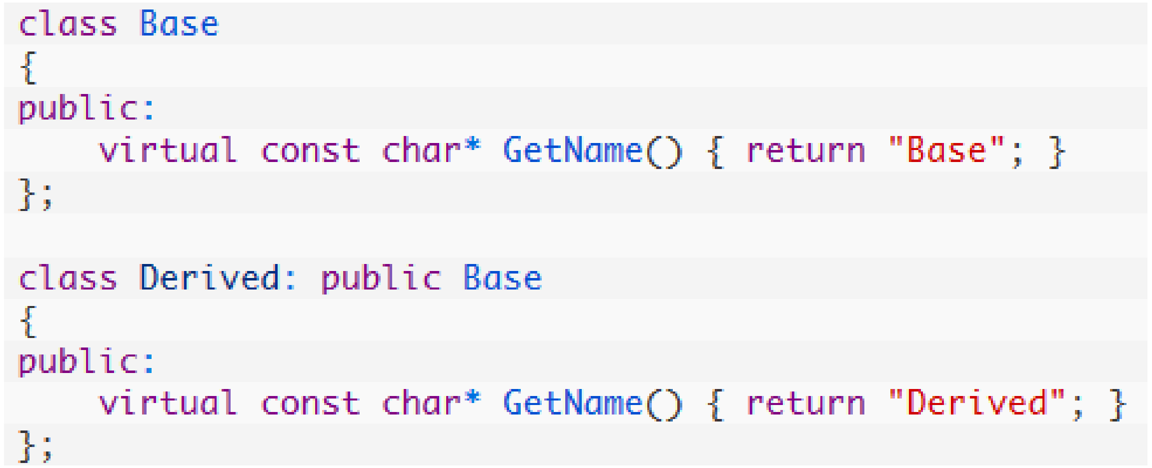
```
int main()
{
    Derived *pDerived = new Derived(5);
    Base *pBase = pDerived;
    delete pBase;

    return 0;
}
```

- Result:

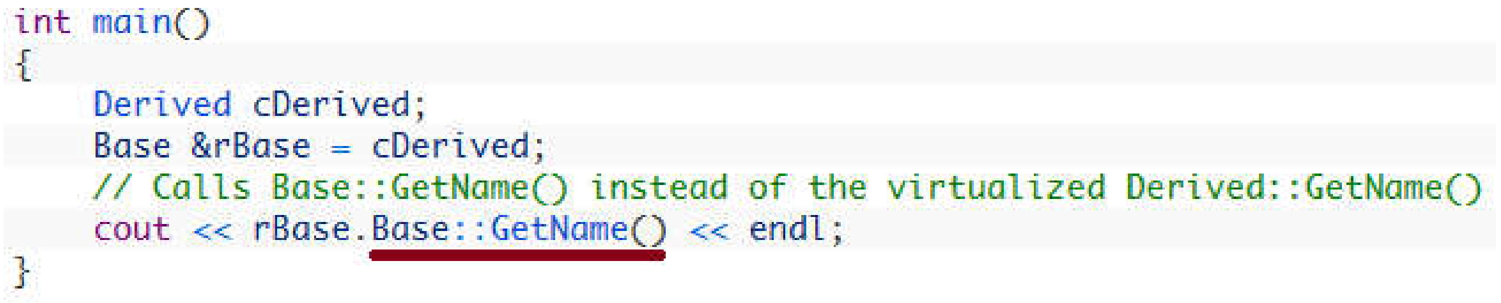
```
Calling ~Derived()
Calling ~Base()
```

Overriding virtualization

- When you want a Base pointer to a Derived object to call Base::GetName() instead of Derived::GetName(), use the scope resolution operator:



```
class Base
{
public:
    virtual const char* GetName() { return "Base"; }
};

class Derived: public Base
{
public:
    virtual const char* GetName() { return "Derived"; }
};
```



```
int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    // Calls Base::GetName() instead of the virtualized Derived::GetName()
    cout << rBase.Base::GetName() << endl;
}
```

The downside of virtual functions

- Why not just make all functions virtual?

- The answer is because it’s **inefficient**:

  - Resolving a virtual function call takes **longer** than a resolving a regular one.

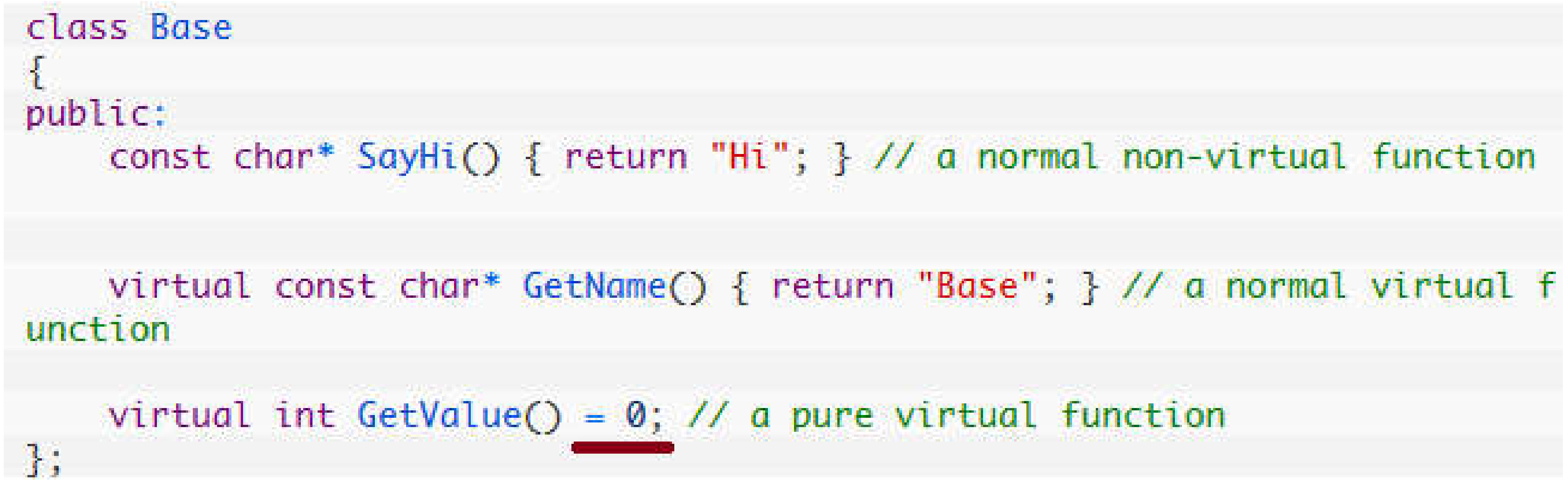
  - Furthermore, the compiler also has to allocate an **extra pointer** for each class object that has one or more virtual functions.

Pure virtual (abstract) functions

- **Pure virtual function** (or **abstract function**) that has no body.

- Simply acts as a placeholder that is meant to be redefined by derived classes.

- To create a pure virtual function, rather than define a body for the function, we simply **assign the function the value 0**.

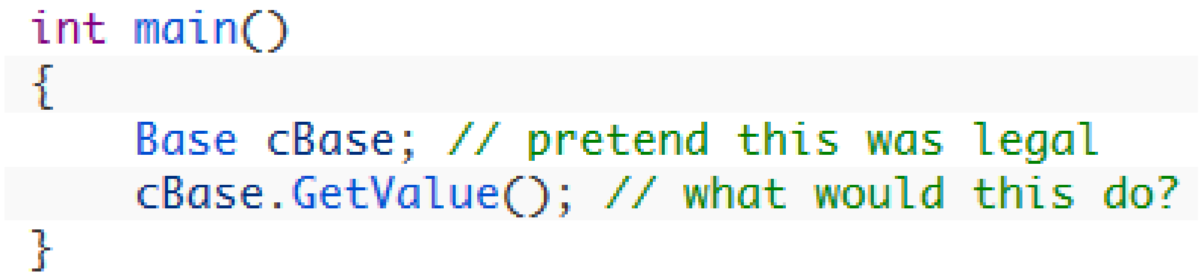


```
class Base
{
public:
    const char* SayHi() { return "Hi"; } // a normal non-virtual function


    virtual const char* GetName() { return "Base"; } // a normal virtual f
unction

    virtual int GetValue() = 0; // a pure virtual function
};
```

Abstract base class

- Any class with one or more pure virtual functions becomes an **abstract base class**, which means that it can not be instantiated.

- What would happen if we could create an instance of Base:



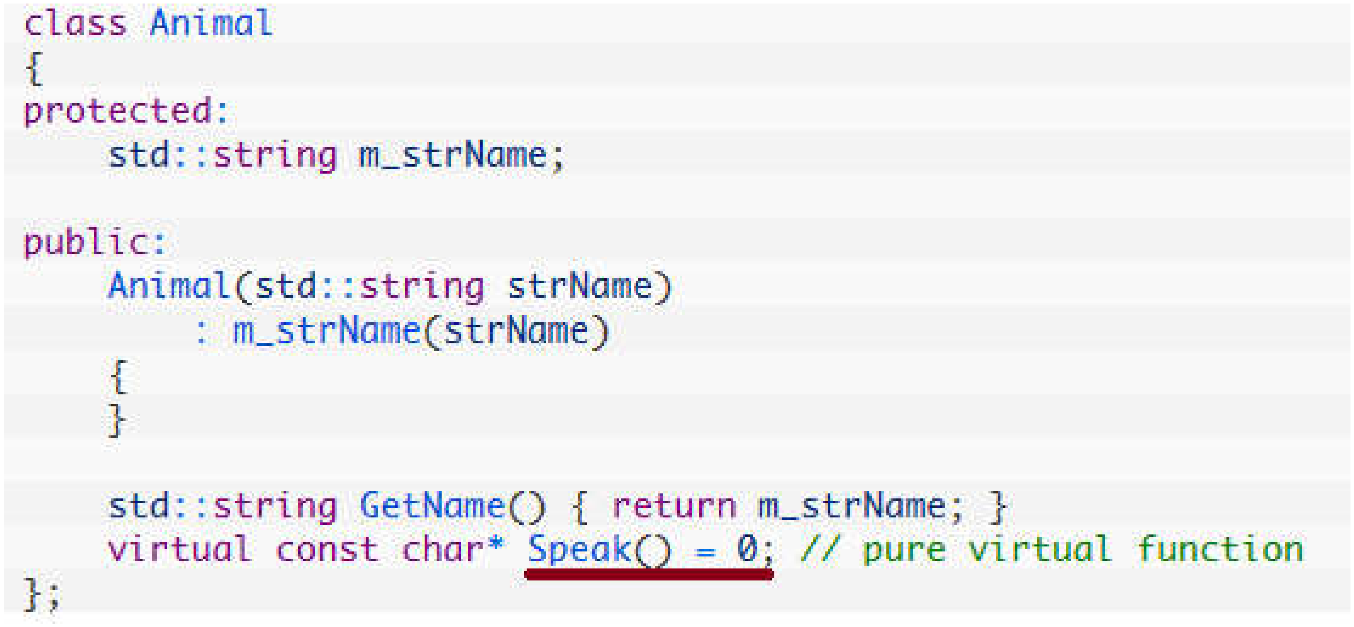
```
int main()
{
    Base cBase; // pretend this was legal
    cBase.GetValue(); // what would this do?
}
```

→ Error when compiling.

- Any derived class must define a body for this function, or that derived class will be considered an abstract base class as well.

Abstract base class

- Make Animal class abstract:

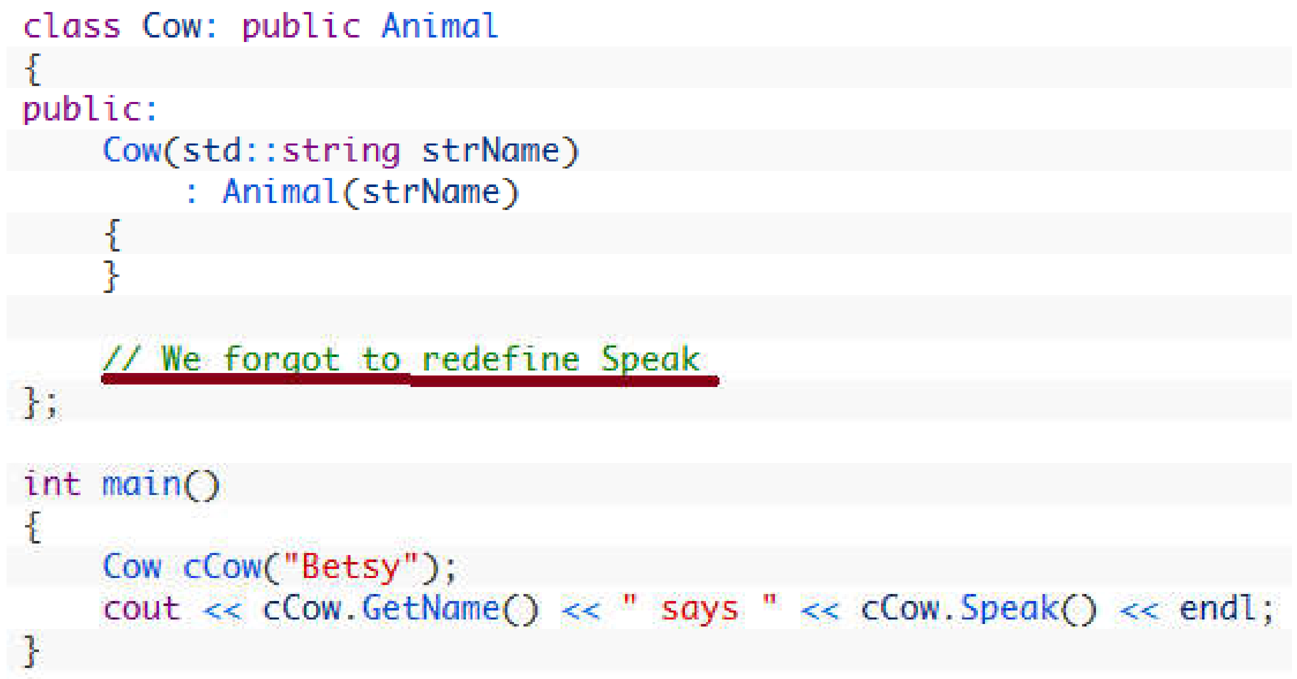


```
class Animal
{
protected:
    std::string m_strName;

public:
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

    std::string GetName() { return m_strName; }
    virtual const char* Speak() = 0; // pure virtual function
};
```

→ Speak() is now a pure virtual function. This means Animal is an **abstract base class**, and **can not be instantiated**.
→ We do **not need to make the constructor protected** any longer to prevent people from allocating objects of type Animal.

Abstract base class

- What happen when Cow class is a derived class and we did not define Cow::Speak(), Cow is also an abstract base class:



```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    // We forgot to redefine Speak
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

→ The compiler will give us an error because Cow is an abstract base class and we can not create instances of abstract base classes.

→ This remind us don’t forget **provides a body for Speak()**.

Abstract base class

A pure virtual function is useful when:

- We have a function that we want to put in the base class, but only let derived classes know what it should return.

- We want the **base class can not be instantiated**.

- The derived classes are forced to define these function before they can be instantiated. This helps ensure the derived classes **do not forget to redefine functions** that the base class was expecting them to.

Interface classes

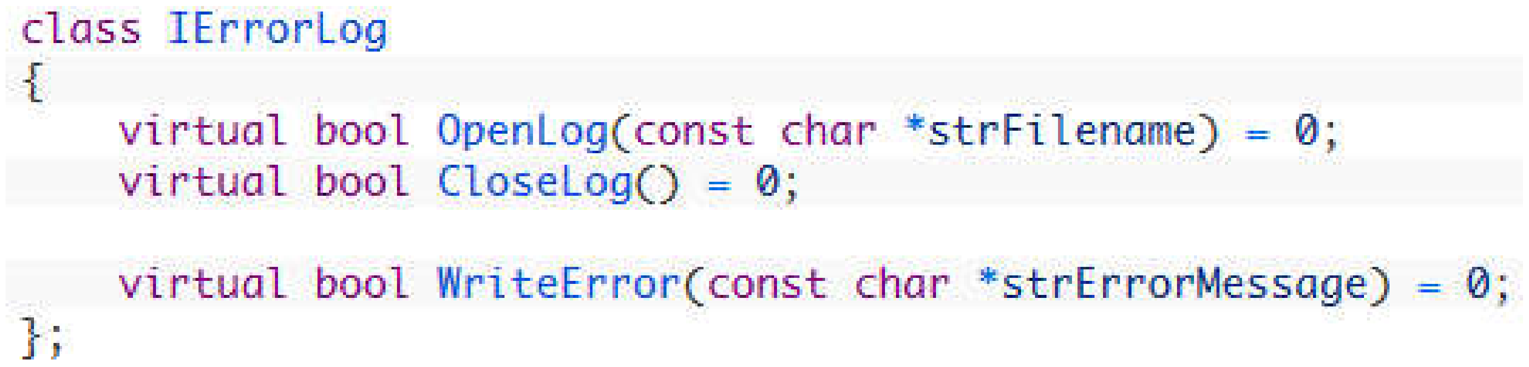
- An **interface class** is a class that has **no members variables** and **all of the functions are pure virtual**.

- In other words, the class is purely a definition, and has no actual implementation.

- Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

Interface classes

- An example of an **interface class:**



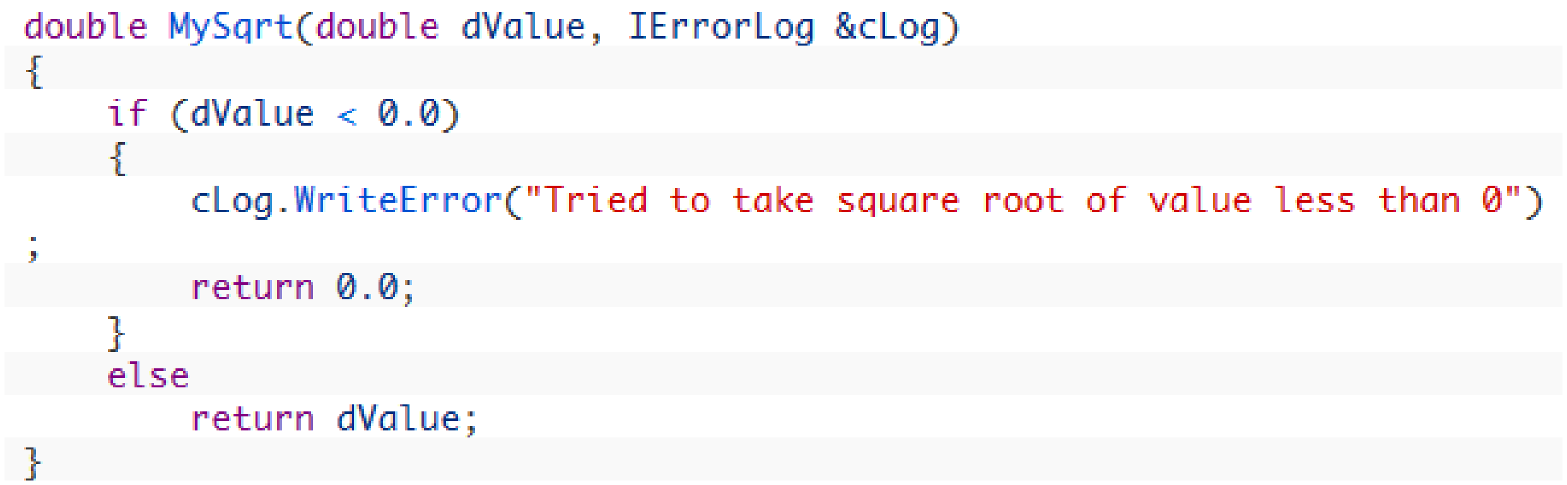
```
class IErrorLog
{
    virtual bool OpenLog(const char *strFilename) = 0;
    virtual bool CloseLog() = 0;

    virtual bool WriteError(const char *strErrorMessage) = 0;
};
```

- Any class inheriting from IErrorLog must provide implementations for all three functions in order to be instantiated.

- You could create derived classes named:
  - **FileErrorLog**, where OpenLog() opens a file on disk, CloseLog() closes it, and WriteError() writes the message to the file.
  - **ScreenErrorLog**, where OpenLog() and CloseLog() do nothing, and WriteError() prints the message in a pop-up message box on the screen.

Interface classes

- Now, let’s say you need to write some code that uses an error log:



```
double MySqrt(double dValue, IErrorLog &cLog)
{
    if (dValue < 0.0)
    {
        cLog.WriteError("Tried to take square root of value less than 0")
;
        return 0.0;
    }
    else
        return dValue;
}
```

- You don’t need to work with FileErrorLog or ScreenErrorLog directly so that you won’t stuck using that kind of error log.

- Now the caller can pass in any class that conforms to the **IErrorLog** interface: If they want the error to go to a **file**, they can pass in an instance of **FileErrorLog**. If they want it to go to the **screen**, they can pass in an instance of **ScreenErrorLog**.

Interface classes

- Interface classes have become extremely popular because they are easy to **use**, easy to **extend**, and easy to **maintain**.

- Use multiple interfaces rather than multiple inheritance.

- Because interfaces have no data and no function bodies, they avoid a lot of the traditional problems with multiple inheritance while still providing much of the flexibility.

Q&A

Thanks for your attention!