12.1 — Pointers and references to the base class of derived objects

BY ALEX ON JANUARY 29TH, 2008 | LAST MODIFIED BY ALEX ON SEPTEMBER 22ND, 2015

In the previous chapter, you learned all about how to use inheritance to derive new classes from existing classes. In this chapter, we are going to focus on one of the most important and powerful aspects of inheritance -- virtual functions.

But before we discuss what virtual functions are, let's first set the table for why we need them.

In the chapter on **construction of derived classes**, you learned that when you create a derived class, it is composed of multiple parts: one part for each inherited class, and a part for itself.

For example, here's a simple case:

```
class Base
{
protected:
    int m_nValue;

public:
    Base(int nValue)
        : m_nValue(nValue)
    {
    }

    const char* GetName() { return "Base"; }
    int GetValue() { return m_nValue; }
};

class Derived: public Base
{
public:
    Derived(int nValue)
        : Base(nValue)
    {
    }

    const char* GetName() { return "Derived"; }
    int GetValueDoubled() { return m_nValue * 2; }
};
```

When we create a Derived object, it contains a Base part (which is constructed first), and a Derived part (which is constructed second). Remember that inheritance implies an is-a relationship between two classes. Since a Derived is-a Base, it is appropriate that Derived contain a Base part of it.

Pointers, references, and derived classes

It should be fairly intuitive that we can set Derived pointers and references to Derived objects:

```
int main()
{
    using namespace std;
    Derived cDerived(5);
    cout << "cDerived is a " << cDerived.GetName() << " and has value " << cDerived.GetValue()
<< endl;

    Derived &rDerived = cDerived;
    cout << "rDerived is a " << rDerived.GetName() << " and has value " << rDerived.GetValue()
<< endl;

    Derived *pDerived = &cDerived;
    cout << "pDerived is a " << pDerived->GetName() << " and has value " << pDerived->GetValue
() << endl;
```

```
    return 0;
}
```

This produces the following output:

```
cDerived is a Derived and has value 5
rDerived is a Derived and has value 5
pDerived is a Derived and has value 5
```

However, since Derived has a Base part, a more interesting question is whether C++ will let us set a Base pointer or reference to a Derived object. It turns out, we can!

```
int main()
{
    using namespace std;
    Derived cDerived(5);

    // These are both legal!
    Base &rBase = cDerived;
    Base *pBase = &cDerived;

    cout << "cDerived is a " << cDerived.GetName() << " and has value " << cDerived.GetValue()
<< endl;
    cout << "rBase is a " << rBase.GetName() << " and has value " << rBase.GetValue() << endl;
    cout << "pBase is a " << pBase->GetName() << " and has value " << pBase->GetValue() << end
l;

    return 0;
}
```

This produces the result:

```
cDerived is a Derived and has value 5
rBase is a Base and has value 5
pBase is a Base and has value 5
```

This result may not be quite what you were expecting at first!

It turns out that because rBase and pBase are a Base reference and pointer, they can only see members of Base (or any classes that Base inherited). So even though Derived::GetName() is an override of Base::GetName(), the Base pointer/reference can not see Derived::GetName(). Consequently, they call Base::GetName(), which is why rBase and pBase report that they are a Base rather than a Derived.

Note that this also means it is not possible to call Derived::GetValueDoubled() using rBase or pBase. They are unable to see anything in Derived.

Here's another slightly more complex example that we'll build on in the next lesson:

```
#include <string>
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 "???"; }
};

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

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

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

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

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;
}
```

This produces the 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 ???
```

We see the same issue here. 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.

Use for pointers and references to base classes

Now you might be saying, “The above examples seem kind of silly. Why would I set a pointer or reference to the base class of a derived object when I can just use the derived object?” It turns out that there are quite a few good reasons.

First, let’s say you wanted to write a function that printed an animal’s name and sound. Without using a pointer to a base class, you’d have to write it like this:

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

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

Not too difficult, but consider what would happen if we had 30 different animal types instead of 2. You'd have to write 30 almost identical functions! Plus, if you ever added a new type of animal, you'd have to write a new function for that one too. This is a huge waste of time considering the only real difference is the type of the parameter.

However, because Cat and Dog are derived from Animal, Cat and Dog have an Animal part. Therefore, it makes sense that we should be able to do something like this:

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

This would let us pass in any class derived from Animal, even ones that we haven't thought of yet! Instead of one function per animal, we get one function that works with all classes derived from Animal!

The problem, is of course, that because cAnimal is an Animal reference, `cAnimal.Speak()` will call Animal::Speak() instead of the derived version of Speak().

Second, let's say you had 3 cats and 3 dogs that you wanted to keep in an array for easy access. Because arrays can only hold objects of one type, without a pointer or reference to a base class, you'd have to do it like this:

```
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() << endl;

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

Now, consider what would happen if you had 30 different types of animals. You'd need 30 arrays, one for each type of animal!

However, 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 Cat and Dog objects
Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZeke };
for (int iii=0; iii < 6; iii++)
    cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Speak() << endl;
```

While this compiles and executes, unfortunately the fact that apcAnimals is a pointer to an Animal means that `apcAnimals[iii]->Speak()` will call Animal::Speak() instead of the proper derived class version of Speak().

Although both of these techniques could save us a lot of time and energy, they have the same problem. The pointer or reference to the base class calls the base version of the function rather than the derived version. If only there was some way to make those base pointers call the derived version of a function instead of the base version…

Want to take a guess what virtual functions are for? 🙂