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

learncpp.com/cpp-tutorial/pointers-and-references-to-the-base-class-of-derived-objects/

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:

```
#include <string_view>

class Base
{
protected:
    int m_value {};

public:
    Base(int value)
        : m_value{ value }
    {
    }

    std::string_view getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

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

    std::string_view getName() const { return "Derived"; }
    int getValueDoubled() const { return m_value * 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.

Pointers, references, and derived classes

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

```
#include <iostream>

int main()
{
    Derived derived{ 5 };
    std::cout << "derived is a " << derived.getName() << " and has value " <<
derived.getValue() << '\n';

    Derived& rDerived{ derived };
    std::cout << "rDerived is a " << rDerived.getName() << " and has value " <<
rDerived.getValue() << '\n';

    Derived* pDerived{ &derived };
    std::cout << "pDerived is a " << pDerived->getName() << " and has value " <<
pDerived->getValue() << '\n';

    return 0;
}
```

This produces the following output:

```
derived 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!

```
#include <iostream>

int main()
{
    Derived derived{ 5 };

    // These are both legal!
    Base& rBase{ derived };
    Base* pBase{ &derived };

    std::cout << "derived is a " << derived.getName() << " and has value " <<
derived.getValue() << '\n';
    std::cout << "rBase is a " << rBase.getName() << " and has value " <<
rBase.getValue() << '\n';
    std::cout << "pBase is a " << pBase->getName() << " and has value " << pBase-
>getValue() << '\n';

    return 0;
}
```

This produces the result:

```
derived 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() shadows (hides) Base::getName() for Derived objects, 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 <iostream>
#include <string_view>
#include <string>

class Animal
{
protected:
    std::string m_name;

    // 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_view name)
        : m_name{ name }
    {
    }

    // To prevent slicing (covered later)
    Animal(const Animal&) = default;
    Animal& operator=(const Animal&) = default;

public:
    std::string_view getName() const { return m_name; }
    std::string_view speak() const { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const { return "Woof"; }
};

int main()
{
    const Cat cat{ "Fred" };
    std::cout << "cat is named " << cat.getName() << ", and it says " << cat.speak()
<< '\n';

    const Dog dog{ "Garbo" };
    std::cout << "dog is named " << dog.getName() << ", and it says " << dog.speak()
<< '\n';

    const Animal* pAnimal{ &cat };
    std::cout << "pAnimal is named " << pAnimal->getName() << ", and it says " <<
pAnimal->speak() << '\n';

    pAnimal = &dog;
    std::cout << "pAnimal is named " << pAnimal->getName() << ", and it says " <<
pAnimal->speak() << '\n';

    return 0;
}
```

This produces the result:

```
cat is named Fred, and it says Meow
dog 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 portion of the 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 using overloaded functions, like this:

```
void report(const Cat& cat)
{
    std::cout << cat.getName() << " says " << cat.speak() << '\n';
}

void report(const Dog& dog)
{
    std::cout << dog.getName() << " says " << dog.speak() << '\n';
}
```

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(const Animal& rAnimal)
{
    std::cout << rAnimal.getName() << " says " << rAnimal.speak() << '\n';
}
```

This would let us pass in any class derived from Animal, even ones that we created after we wrote the function! Instead of one function per derived class, we get one function that works with all classes derived from Animal!

The problem is, of course, that because rAnimal is an Animal reference, `rAnimal.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 create a different array for each derived type, like this:

```
#include <array>
#include <iostream>

// Cat and Dog from the example above

int main()
{
    const auto& cats{ std::to_array<Cat>({{ "Fred" }, { "Misty" }, { "Zeke" }}) };
    const auto& dogs{ std::to_array<Dog>({{ "Garbo" }, { "Pooky" }, { "Truffle" }})
};

    // Before C++20
    // const std::array<Cat, 3> cats{{ { "Fred" }, { "Misty" }, { "Zeke" } }};
    // const std::array<Dog, 3> dogs{{ { "Garbo" }, { "Pooky" }, { "Truffle" } }};

    for (const auto& cat : cats)
    {
        std::cout << cat.getName() << " says " << cat.speak() << '\n';
    }

    for (const auto& dog : dogs)
    {
        std::cout << dog.getName() << " says " << dog.speak() << '\n';
    }

    return 0;
}
```

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 derived from Animal, it makes sense that we should be able to do something like this:

```
#include <array>
#include <iostream>

// Cat and Dog from the example above

int main()
{
    const Cat fred{ "Fred" };
    const Cat misty{ "Misty" };
    const Cat zeke{ "Zeke" };

    const Dog garbo{ "Garbo" };
    const Dog pooky{ "Pooky" };
    const Dog truffle{ "Truffle" };

    // Set up an array of pointers to animals, and set those pointers to our Cat and
Dog objects
    const auto animals{ std::to_array<const Animal*>({&fred, &garbo, &misty, &pooky,
&truffle, &zeke }) };

    // Before C++20, with the array size being explicitly specified
    // const std::array<const Animal*, 6> animals{ &fred, &garbo, &misty, &pooky,
&truffle, &zeke };

    for (const auto animal : animals)
    {
        std::cout << animal->getName() << " says " << animal->speak() << '\n';
    }

    return 0;
}
```

While this compiles and executes, unfortunately the fact that each element of array “animals” is a pointer to an Animal means that `animal->speak()` will call Animal::speak() instead of the derived class version of speak() that we want. The output is

```
Fred says ???
Garbo says ???
Misty says ???
Pooky says ???
Truffle says ???
Zeke says ???
```

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? :)

Quiz time

1. Our Animal/Cat/Dog example above doesn’t work like we want because a reference or pointer to an Animal can’t access the derived version of speak() needed to return the right value for the Cat or Dog. One way to work around this issue would be to make the data returned by the speak() function accessible as part of the Animal base class (much like the Animal’s name is accessible via member m_name).

Update the Animal, Cat, and Dog classes in the lesson above by adding a new member to Animal named m_speak. Initialize it appropriately. The following program should work properly:

```
#include <array>
#include <iostream>

int main()
{
    const Cat fred{ "Fred" };
    const Cat misty{ "Misty" };
    const Cat zeke{ "Zeke" };

    const Dog garbo{ "Garbo" };
    const Dog pooky{ "Pooky" };
    const Dog truffle{ "Truffle" };

    // Set up an array of pointers to animals, and set those pointers to our Cat and
Dog objects
    const auto animals{ std::to_array<const Animal*>({ &fred, &garbo, &misty, &pooky,
&truffle, &zeke }) };

    // Before C++20, with the array size being explicitly specified
    // const std::array<const Animal*, 6> animals{ &fred, &garbo, &misty, &pooky,
&truffle, &zeke };

    for (const auto animal : animals)
    {
        std::cout << animal->getName() << " says " << animal->speak() << '\n';
    }

    return 0;
}
```

Show Solution

2. Why is the above solution non-optimal?

Hint: Think about the future state of Cat and Dog where we want to differentiate Cats and Dogs in more ways.
Hint: Think about the ways in which having a member that needs to be set at initialization

limits you.

Show Solution