12.8 — Object slicing

BY ALEX ON NOVEMBER 19TH, 2016 | LAST MODIFIED BY ALEX ON MAY 23RD, 2017

Let's go back to an example we looked at previously:

```
class Base
{
protected:
    int m_value;

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

    virtual const char* getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

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

    virtual const char* getName() const { return "Derived"; }
};

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

    Base &ref = derived;
    std::cout << "ref is a " << ref.getName() << " and has value " << ref.getValue() << '\n';

    Base *ptr = &derived;
    std::cout << "ptr is a " << ptr->getName() << " and has value " << ptr->getValue() << 'n';

    return 0;
}
```

In the above example, ref references and ptr points to derived, which has a Base part, and a Derived part. Because ref and ptr are of type Base, ref and ptr can only see the Base part of derived -- the Derived part of derived still exists, but simply can't be seen through ref or ptr. However, through use of virtual functions, we can access the most-derived version of a function. Consequently, the above program prints:

```
derived is a Derived and has value 5
ref is a Derived and has value 5
ptr is a Derived and has value 5
```

But what happens if instead of setting a Base reference or pointer to a Derived object, we simply *assign* a Derived object to a Base object?

```
int main()
{
    Derived derived(5);
    Base base = derived; // what happens here?
    std::cout << "base is a " << base.getName() << " and has value " << base.getValue() <<
'\n';

    return 0;
}
```

Remember that derived has a Base part and a Derived part. When we assign a Derived object to a Base object, only the Base portion of the Derived object is copied. The Derived portion is not. In the example above, base receives a copy of the Base portion of derived, but not the Derived portion. That Derived portion has effectively been "sliced off". Consequently, the assigning of a Derived class object to a Base class object is called **object slicing** (or slicing for short).

Because variable base does not have a Derived part, base.getName() resolves to Base::getName().

The above example prints:

```
base is a Base and has value 5
```

Used conscientiously, slicing can be benign. However, used improperly, slicing can cause unexpected results in quite a few different ways. Let's examine some of those cases.

Slicing and functions

Now, you might think the above example is a bit silly. After all, why would you assign derived to base like that? You probably wouldn't. However, slicing is much more likely to occur accidentally with functions.

Consider the following function:

```
void printName(const Base base) // note: base passed by value, not reference
{
    std::cout << "I am a " << base.getName() << '\n';
}
```

This is a pretty simple function with a const base object parameter that is passed by value. If we call this function like such:

```
int main()
{
    Derived d(5);
    printName(d); // oops, didn't realize this was pass by value on the calling end

    return 0;
}
```

When you wrote this program, you may not have noticed that base is a value parameter, not a reference. Therefore, when called as printName(d), we might have expected base.getName() to call virtualized function getName() and print "I am a Derived", that is not what happens. Instead, Derived object d is sliced and only the Base portion is copied into the base parameter. When base.getName() executes, even though the getName() function is virtualized, there's no Derived portion of the class for it to resolve to. Consequently, this program prints:

```
I am a Base
```

In this case, it's pretty obvious what happened, but if your functions don't actually print any identifying information like this, tracking down the error can be challenging.

Of course, slicing here can all be easily avoided by making the function parameter a reference instead of a pass by value (yet another reason why passing classes by reference instead of value is a good idea).

```
void printName(const Base &base) // note: base now passed by reference
{
    std::cout << "I am a " << base.getName() << '\n';
}

int main()
{
    Derived d(5);
    printName(d);

    return 0;
}
```

This prints:

```
I am a Derived
```

Slicing vectors

Yet another area where new programmers run into trouble with slicing is trying to implement polymorphism with std::vector. Consider the following program:

```
#include <vector>
int main()
{
    std::vector<Base> v;
    v.push_back(Base(5)); // add a Base object to our vector
    v.push_back(Derived(6)); // add a Derived object to our vector

        // Print out all of the elements in our vector
    for (int count = 0; count < v.size(); ++count)
        std::cout << "I am a " << v[count].getName() << " with value " << v[count].getValue()
 << "\n";

    return 0;
}
```

This program compiles just fine. But when run, it prints:

```
I am a Base with value 5
I am a Base with value 6
```

Similar to the previous examples, because the std::vector was declared to be a vector of type Base, when Derived(6) was added to the vector, it was sliced.

Fixing this is a little more difficult. Many new programmers try creating a std::vector of references to an object, like this:

```
std::vector<Base&> v;
```

Unfortunately, this won't compile. The elements of std::vector must be assignable, whereas references can't be reassigned (only initialized).

One way to address this is to make a vector of pointers:

```
#include <vector>
int main()
{
```

```
    std::vector<Base*> v;
    v.push_back(new Base(5)); // add a Base object to our vector
    v.push_back(new Derived(6)); // add a Derived object to our vector

        // Print out all of the elements in our vector
    for (int count = 0; count < v.size(); ++count)
        std::cout << "I am a " << v[count]->getName() << " with value " << v[count]->getValue() << "\n";

    for (int count = 0; count < v.size(); ++count)
        delete v[count];

    return 0;
}
```

This prints:

```
I am a Base with value 5
I am a Derived with value 6
```

which works! But it's quite a bit of additional headache since you now have to deal with dynamic memory allocation.

There's one other way to resolve this. The standard library provides a useful workaround: the std::reference_wrapper class. Essentially, std::reference_wrapper is a class that acts like a reference, but also allows assignment and copying, so it's compatible with std::vector.

The good news is that you don't really need to understand how it works to use it. All you need to know are three things:
1) std::reference_wrapper lives in the <functional> header
2) When you create your std::reference_wrapper wrapped object, the object can't be an anonymous object (since anonymous objects have expression scope would leave the reference dangling)
3) When you want to get your object back out of std::reference_wrapper, you use the get() member function.

Here's our code rewritten to use std::reference_wrapper:

```
#include <vector>
#include <functional> // for std::reference_wrapper
int main()
{
    std::vector<std::reference_wrapper<Base> > v; // our vector is a vector of std::reference_wrapper wrapped Base (not Base&)
    Base b(5); // b and d can't be anonymous objects
    Derived d(6);
    v.push_back(b); // add a Base object to our vector
    v.push_back(d); // add a Derived object to our vector

    // Print out all of the elements in our vector
    for (int count = 0; count < v.size(); ++count)
        std::cout << "I am a " << v[count].get().getName() << " with value " << v[count].get().getValue() << "\n"; // we use .get() to get our element from the wrapper


    return 0;
}
```

This works as you'd expect:

```
I am a Base with value 5
I am a Derived with value 6
```

and avoids having to deal with dynamic memory.

If this seems a bit obtuse or obscure at this point (especially the nested types), come back to it later after we've covered template classes and you'll likely find it more understandable.

The Frankenobject

In the above examples, we've seen cases where slicing lead to the wrong result because the derived class had been sliced off. Now let's take a look at another dangerous case where the derived object still exists!

Consider the following code:

```
1  int main()
2  {
3      Derived d1(5);
4      Derived d2(6);
5      Base &b = d2;
6
7      b = d1; // this line is problematic
8
9      return 0;
10 }
```

The first three lines in the function are pretty straightforward. Create two Derived objects, and set a Base reference to the second one.

The fourth line is where things go astray. Since b points at d2, and we're assigning d1 to b, you might think that the result would be that d1 would get copied into d2 -- and it would, if b were a Derived. But b is a Base, and the operator= that C++ provides for classes isn't virtual by default. Consequently, only the Base portion of d1 is copied into d2.

As a result, you'll discover that d2 now has the Base portion of d1 and the Derived portion of d2. In this particular example, that's not a problem (because the Derived class has no data of its own), but in most cases, you'll have just created a Frankenobject -- composed of parts of multiple objects. Worse, there's no easy way to prevent this from happening (other than avoiding assignments like this as much as possible).

Conclusion

Although C++ supports assigning derived objects to base objects via object slicing, in general, this is likely to cause nothing but headaches, and you should generally try to avoid slicing. Make sure your function parameters are references (or pointers) and try to avoid any kind of pass-by-value when it comes to derived classes.


12.9 -- Dynamic casting


Index


12.7 -- Virtual base classes

27 comments to 12.8 — Object slicing

Akshay
November 6, 2017 at 8:49 pm · Reply

Hi Alex,
While copying derived object to base object we know that only base portion of the derived object gets copied to new base object, but shouldn't that also copy __vptr__ which is pointing to derived's vtable be copied as is into the new base object? if not the case, How and when is __vtpr__'s value is changed while slicing happens?

Thanks, for the great tutorial!

Luhan
November 8, 2017 at 6:45 am · Reply

He said in the comments "Virtual tables are set up per class, not per object, so they are not affected when you copy individual objects.". What happens is __vptr__ is created in the Base, and inherited by the Derived classes, which utilizes the __vptr__ inherited to point to their own vtable. At this part Base have access to the __vptr__ , because it's a member of Base. But when you copy (slice the object), the derived part isn't more present, as Alex said "With a pointer or reference, you're not making a copy -- you're passing a way of indirectly accessing the object". Because you are slicing the object, the Base class doesn't have more access to the __vptr__ of Derived, which points to the vtable of Derived.

In resume, the Derived part because of the slicing doesn't exist in that object (I mean the Base object receiving the copy), because when doing assignment, Alex did say that there isn't a default operator= to handle it right, so it's copying only the mutual part these 2 variables have, which is the Base portion of the class.

Correct me if I'm wrong Alex.

Alex
November 8, 2017 at 8:47 pm · Reply

The virtual pointer isn't a normal member variable, so it's not copied. The virtual pointer in the sliced base class is set (to the base class virtual table) when the sliced base object is created.

AMG
October 24, 2017 at 8:37 pm · Reply

Alex,
I thought std::vector is supposed to contain entities of the same type, but Base and Derived are different, and so pointers to Base and Derived. Why v.push_back(Derived(6)) (or v.push_back(new Derived(6)) is allowed? Thank you.

Alex
October 25, 2017 at 12:23 pm · Reply

You've always been able to assign values of one type to a vector of a different element type, so long as the value can be converted to the vector's element type.

Since Derived pointers (or references) can be converted to Base pointers (or references), this is fine -- the value is simply converted to the vector's element type and stored.

Panagiotis
May 23, 2017 at 12:25 am · Reply

In the section "Slicing and functions" our function's parameter is const Base &base. So, as we have already learnt, the compiler won't let us call any functions that do not have the "const" keyword. I think that our function's declaration should have been like

```
1 virtual const char* getName() const { return "Base"; }
```

for Base class, and

```
1 virtual const char* getName() const { return "Derived"; }
```

for Derived class

Alex
May 23, 2017 at 2:33 pm · Reply

Good catch. I've updated the member functions to be const. Thanks!

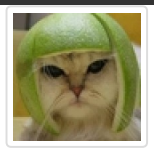
nagarajsherigar
April 20, 2017 at 4:10 am · Reply

Base base = derived;
Assuming assignment operator is called
wont derived *__vptr copied to base *__vptr?

nagarajsherigar
April 20, 2017 at 4:26 am · Reply

That means *__vptr is not assigned. It is only pointer to vtable when the object is created. Since base object is created here

Alex
April 20, 2017 at 1:38 pm · Reply



No. Virtual tables are set up per class, not per object, so they are not affected when you copy individual objects.

nagarajsherigar
April 20, 2017 at 7:48 pm · Reply

hi Alex

Thanks for the reply

Joe
April 9, 2017 at 2:26 am · Reply

Hi Alex,

Thanks a lot for your tutorials, they are tremendously helpful!

I have a question which relates to both the concept of virtual functions and the object slicing. Consider the following classes:

```
1  class Animal
2  {
3  protected:
4      std::string m_name;
5  public:
6      Animal(std::string name)
7          : m_name(name)
8      {
9      }
10
11     virtual const char* getName() const {return "Animal";}
12 };
13
14 class Cow : public Animal
15 {
16 private:
17     int m_litresOfMilkPerDay;
18
19 public:
20     Cow(std::string name, int milk)
21         : Animal(name), m_litresOfMilkPerDay(milk)
22     {
23     }
24
25     virtual const char* getName() const {return "Cow";}
26     int getLitresOfMilkPerDay() const {return m_litresOfMilkPerDay;} // This function is i
   ntrinsic to Cow
27 };
28
29 class Chicken : public Animal
30 {
31 private:
32     int m_numberOfEggsPerDay;
33
34 public:
35     Chicken(std::string name, int eggs)
36         : Animal(name), m_numberOfEggsPerDay(eggs)
37     {
```

```
38      }
39
40      virtual const char* getName() const {return "Chicken";}
41      int getNumberOfEggsPerDay() const {return m_numberOfEggsPerDay;}  // This function is
     intrinsic to Chicken
42  };
43
44  class Stable
45  {
46  private:
47      std::vector<std::reference_wrapper<Animal> > m_animalsInStable;
48
49  public:
50      Stable() {}
51      void addAnimal(Animal& animal)
52      {
53          m_animalsInStable.push_back(animal);
54      }
55  };
```

If I put animals of any flavour into m_animalsInStable, is there a way to call the intrinsic functions other than declaring them virtual in the Animal base class? To put it differently, within a member function of Stable can I do something like m_animalsInStable[0].get().getLitresOfMilkPerDay() ?

Alex
April 10, 2017 at 11:58 am · Reply

You have two main options:
1) Add a virtual function to the Animal base class, call that, and let it virtually resolve to the most derived class. This generally only makes sense for functions that apply to all Animals, but it's nice because you don't have to know what all the various derived classes are up-front.
2) Dynamically cast your Animal into a derived object and call whatever function you want directly. The tricky part here is that you have to have some way to determine what derived class the Animal actually is.

You _could_ create a virtual get() function that returns a pointer (or reference) to the derived object (so Cow would return a Cow* or Cow& to this or *this), but it doesn't really buy you anything that you can't do with a dynamic_cast anyway.

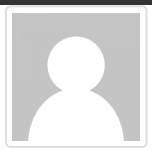
One way to solve #2 would be to create an AnimalType enum. Then create a virtual getAnimalType() function in Animal. Each class can self-identify by returning the proper enum. You can switch on this enum to do whatever you need to do with a derived-class-specific function. e.g.

```
1  switch (m_animalsInStable[0].getAnimalType)
2  {
3  case ANIMALS_COW:
4      {
5      std::cout << dynamic_cast<Cow>(m_animalsInStable[0]).getLitresOfMilkPerDay();
6      }
7  }
```

Joe
April 10, 2017 at 12:10 pm · Reply

Thanks, your answer is much appreciated!

Hugh Mungus
April 5, 2017 at 7:42 am · Reply



Hey Alex,

Maybe I missed something, but can you explain why passing an object by value causes slicing and passing by reference doesn't?

Alex
April 5, 2017 at 12:44 pm · Reply

When you pass an object by value, a copy of the object is made. The object's type is used to determine what to copy. Thus, if your type is a base type, only the base part will be copied.

With a pointer or reference, you're not making a copy -- you're passing a way of indirectly accessing the object. That indirection might only see the base part of the object, but the rest of the object is still there.

Hugh Mungus
April 5, 2017 at 5:51 pm · Reply

Ah i see, thank you

Gary Wong
June 9, 2017 at 7:05 pm · Reply

Alex, thank you for your explanation. However, I am still a bit confused.
QUOTED: "Thus, if your type is a base type, only the base part will be copied."

*__vptr is part of base type, will the *__vptr of Derived be copied to the Base object?
If yes, then the *__vptr of Base Object should point to the function of Derived?

Then why the function of Base is called eventually? Unless there is mechanism of degrading the function call to the Base Class (meaning if the *__vptr is pointing to Derived function, but as there is no Derived function in the Base object, it resolves to Base function)…?

I really wish that you could help clear up my confusion. Much appreciated!

Alex
June 9, 2017 at 8:35 pm · Reply

> *__vptr is part of base type, will the *__vptr of Derived be copied to the Base object?

No. The Base class vptr is used, which only points to Base functions.

loveu
March 31, 2017 at 6:41 am · Reply

Hi alex thanks again for replying so efficiently to my questions!

```
#include <vector>
int main()
{
    std::vector<Base*> v;
    v.push_back(new Base(5)); // add a Base object to our vector
    v.push_back(new Derived(6)); // add a Derived object to our vector

        // Print out all of the elements in our vector
    for (int count = 0; count < v.size(); ++count)
```

```
        std::cout << "I am a " << v[count]->getName() << " with value " << v[count]->getValue() << "\n";

    for (int count = 0; count < v.size(); ++count)
        delete v[count];

    return 0;
}
```

for this example, std::vector creates an array of pointers which point to dynamically allocated pointers which point to dynamically allocated objects am I right? So the std::vector deals with the deallocation of dynamically allocated pointers and we have to seperately deal with dynamically allocated objects as you have done so with

```
for (int count = 0; count < v.size(); ++count)
        delete v[count];

    return 0;
```

am I going in the right direction?

Alex
April 2, 2017 at 12:29 pm · Reply

Yes, you have it correct. There's just a typo in your sentence, "std::vector creates an array of pointers which point to dynamically allocated pointers which point to dynamically allocated objects am I right"

It should be: std::vector creates an array of pointers, each of which point to dynamically allocated objects.

David
March 2, 2017 at 12:02 am · Reply

You explained Franken object by below lines:

Derived d1(5);
Derived d2(6);
Base &b = d2;

b = d1;

Here in line b=d1; Base reference b is getting reassigned. Shouldn't it throw compile error ??

Thanks

Alex
March 2, 2017 at 9:59 am · Reply

References can't be reassigned. The are set upon initialization and thereafter any assignment to them is an assignment to the referenced object, not a reassignment of the reference itself.

So in the quoted example, b = d1 means "assign d1 into the object b is referencing (which is d2)". But since b is a Base, it only copies the Base potion of d1 into d2.

Jonas
December 15, 2016 at 3:45 am · Reply

Hello, Alex! First off, thank you so much for these tutorials! I've been lurking for about a year. Great stuff! My question is what happens if I use pop_back() on the vector<std::reference_wrapper<object>>?

Does it only remove the reference or does it call the object's destructor as well?
And furthermore, if the object was created using new, is there anything else I have to think about?
I tried using delete on both the the wrapper and the object (using .get()) but the compiler wouldn't let me, obviously
Cheers.

Alex
December 15, 2016 at 2:42 pm · Reply

Calling pop_back on a reference wrapped element just removes the reference element from the array, it doesn't call any destructors. If the object was created using new, then you'll need to manually destroy it. But if you're creating objects using new, then you should probably be using a std::vector of pointers rather than reference_wrapper.

Jonas
December 16, 2016 at 10:52 am · Reply

Alright, cool. Just wanted to know my options. Thanks a lot!

Saiyu
December 7, 2016 at 3:57 am · Reply

I didn't notice this before, thank the author!