25.9 — Object slicing

ALEX[1] OCTOBER 28, 2024

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

COPY

```cpp
#include <iostream>
#include <string_view>

class Base
{
protected:
    int m_value{};

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

    virtual ~Base() = default;

    virtual 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 override { 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?

```cpp
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 base was and still is just a Base, Base’s virtual pointer still points to Base. Thus, 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:

```cpp
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:

```cpp
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), while 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).

```cpp
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:

```cpp
#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 (const auto& element : v)
        std::cout << "I am a " << element.getName() << " with value " <<
element.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:

```cpp
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:

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<Base*> v{};

    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 (const auto* element : v)
        std::cout << "I am a " << element->getName() << " with value " << element-
>getValue() << '\n';

    return 0;
}
```

This prints:

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

which works! A few comments about this. First, nullptr is now a valid option, which may or may not be desirable. Second, you now have to deal with pointer semantics, which can be awkward. But the upside is that using pointers allows us to put dynamically allocated objects in the vector (just don’t forget to explicitly delete them).

Another option is to use std::reference_wrapper, which is a class that mimics an reassignable reference:

```cpp
#include <functional> // for std::reference_wrapper
#include <iostream>
#include <string_view>
#include <vector>

class Base
{
protected:
    int m_value{};

public:
    Base(int value)
        : m_value{ value }
    {
    }
    virtual ~Base() = default;

    virtual 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 override { return "Derived"; }
};

int main()
{
    std::vector<std::reference_wrapper<Base>> v{}; // a vector of reassignable
references to 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
    // we use .get() to get our element out of the std::reference_wrapper
    for (const auto& element : v) // element has type const
std::reference_wrapper<Base>&
        std::cout << "I am a " << element.get().getName() << " with value " <<
element.get().getValue() << '\n';

    return 0;
}
```

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:

```cpp
int main()
{
    Derived d1{ 5 };
    Derived d2{ 6 };
    Base& b{ d2 };

    b = d1; // this line is problematic

    return 0;
}
```

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, the assignment operator that copies a Base is invoked, and 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).

Tip

If the Base class is not designed to be instantiated by itself (e.g. it is just an interface class), slicing can be avoided by making the Base class non-copyable (by deleting the Base copy constructor and Base assignment operator).

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.

Next lesson
25.10 Dynamic casting

2

Back to table of contents

3

Previous lesson
25.8 Virtual base classes

4

5

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!
Avatars from https://gravatar.com/[7] are connected to your provided email address.

110 COMMENTS Newest

Bhargavi
March 6, 2025 2:33 pm PST

Hi Alex, trying to understand below piece of code

Base& b{ d2 };

b = d1; // this line is problematic

Here I see b as reference to Base class and

b = d1 means we are trying to reassign the reference right? and this doesn't even compile right?

2 Reply

Jeremy
Reply to Bhargavi [8] May 3, 2025 1:59 pm PDT

You are not reassigning the reference. Do a quick review of references.

2 Reply

EmtyC
December 29, 2024 11:12 pm PST

How about move semantics, I guess the effect is the same as copy semantics?
How about virtual move and copy constructors (or non logical, like a virtual constructor)?

0 Reply

Raiden
March 6, 2024 7:21 pm PST

> But on the upside, this also allows the possibility of dynamic memory allocation, which is useful if your objects might otherwise go out of scope.
What does it mean here?

0 Reply

Alex Author
Reply to Raiden [9] March 9, 2024 4:39 pm PST

Since the vector is a vector of pointers, you can set those pointers to point at dynamically allocated objects. Those objects will stay in scope until you explicitly delete them.

0 Reply

D D
December 10, 2023 7:59 am PST

Hello, you should add `virtual dtor` to the Base class

```
lass Base
{
protected:
    int m_value{};

public:
virtual ~Base() = default; //FORGOTTEN
    Base(int value)
        : m_value{ value }
    {
    }

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

2. Because you have `Base{5}`, you can replace `push_back` to `emplace_back` in your examples.

```
std::vector<Base> v{};
v.emplace_back(5); // add a Base object to our vector
v.emplace_back(Derived{ 6 }); // add a Derived object to our vector
```

3. Can you add an article for std::vector and its methods? I think the important ones are difference between `push_back` and `emplace_back`

Last edited 1 year ago by D D

0 Reply

Alex Author
Reply to D D 10 December 11, 2023 5:07 pm PST

See https://www.learncpp.com/cpp-tutorial/stdvector-and-stack-behavior/#push_vs_emplace

0 Reply

O

Onur
December 2, 2023 3:40 am PST

I think this is also a convenient alternative to a vector of raw pointers.

```
int main()
{
    std::vector<std::unique_ptr<Base>> v{}; // a vector of unique_ptr's to Base

    std::unique_ptr<Base> b{ std::make_unique<Base>(5) }; // create a Base object
    std::unique_ptr<Derived> d{ std::make_unique<Derived>(6) }; // create a Derived
object

    v.push_back(std::move(b)); // add a Base object to our vector
    v.push_back(std::move(d)); // add a Derived object to our vector

    // Print out all of the elements in our vector
    for (const auto& element : v) // element has type const std::unique_ptr<Base>&
        std::cout << "I am a " << element->getName() << " with value " << element-
>getValue() << '\n';

    return 0;
}
```

6 Reply

Helix
October 12, 2023 11:26 pm PDT

I was searching for situations where deleting functions would be useful, almost necessary even. Avoiding object slicing is one such situation, where you'd want to delete the copy and move constructors of the base class.
If someone wants to read what Bjarne Stroustrup had to say about this, read here (https://stackoverflow.com/a/45736918)[11].

1 Reply

Cuppar
June 8, 2023 1:03 am PDT

Can we make all base class's operator= to virtual to prevent the Frankenobject problem?
What's the downside about to do this?

0 Reply

> **Alex** Author
> Reply to Cuppar[12] June 11, 2023 11:25 pm PDT
>
> I don't think so. How are you thinking this would work?
>
> 0 Reply
>
> > **Cuppar**
> > Reply to Alex[13] June 12, 2023 12:13 am PDT
> >
> > Oh, yes. It's not work. I understand now.

0 Reply

Emeka Daniel
May 21, 2023 5:13 pm PDT

So, while playing around with it, I found something odd, in public member function std::reference_wrapper::get().
This is how it is defined:

constexpr int& std::reference_wrapper<int>::get() const noexcept

it returns a non const reference to the object it is referencing(obviously, for the user to be able to change the value of the referenced object, because the operator= has already been overloaded for reseating), I get all that, but what buggles me is the const qualifier added, which by default when used, mandates all return types to be either be const reference or value, but clearly it doesn't do that. So I made a protype class to help me understand:

```
class A
{
    int m_int{};
public:
    A(int e): m_int{e}
    {}
    ~A() {}

    int& operator()() const
    {
        // I figured this is how they pulled it off.
        return const_cast<int&>(m_int) ;
    }

    int get() const
    {
        return m_int;
    }

};
```

The only reason i could think of, of why they did such, was to make even const instances of the class eligible to use the function. Correct me if i'm wrong.

Last edited 2 years ago by Emeka Daniel

0 Reply

Strain
Reply to Emeka Daniel 14 April 12, 2024 5:25 am PDT

I think that what the authors had in mind is that if we don't want to reseat the reference, then we make the `std::reference_wrapper`

`const` (or, even better, just use normal references). However, if we don't want to change what's being referenced, we would do `std::reference_wrapper<const T>`.

Something similar happens with `std::unique_ptr` – making the variable `const` still allows you to mutate the pointee, but not change what it is pointing to.

Last edited 1 year ago by Strain

0 Reply

Emeka Daniel

Reply to Strain 15 April 12, 2024 7:42 am PDT

Yh, makes a lot of sense.

0 Reply

Alex Author

Reply to Emeka Daniel 14 May 22, 2023 4:53 pm PDT

It does appear that get() is discarding the const of a const std::reference_wrapper, as it allows you to reseat a const std::reference_wrapper. That is quite strange. I don't understand the use case for this.

1 Reply

Emeka Daniel

Reply to Alex 16 May 23, 2023 12:05 am PDT

Oh, okay. There's no problem anyways, I just wanted to find of it was only me that found it strange.

0 Reply

u262d

April 4, 2023 11:16 pm PDT

Derived d1{ 5 };
Derived d2{ 6 };
Base& b{ d2 };
b = d1; // this line is problematic

in chapter 9.3 we learned that a reference cannot be reseated
"References can’t be reseated (changed to refer to another object)"

or am i missing smthng?

1 Reply

Alex Author

Reply to u262d 17 April 5, 2023 10:03 pm PDT

As you note, references can't be reseated, so hopefully the dev wasn't expecting `b = d1` to reseat `b`.

The issue at play here is that `b` is a `Base`, so `b = d1` only copies the `Base` portion of `d1` into `d2`, not the whole thing.

2 Reply

u262d
April 4, 2023 3:57 am PDT

thanka yooo, new english word learned: benign
"Used conscientiously, slicing can be benign."

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/dynamic-casting/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/
5. https://www.learncpp.com/object-slicing/
6. https://www.learncpp.com/cpp-tutorial/using-declarations-and-using-directives/
7. https://gravatar.com/
8. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-608307
9. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-594374
10. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-590739
11. https://stackoverflow.com/a/45736918
12. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-581460
13. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-581693
14. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-580625
15. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-595681
16. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-580678
17. https://www.learncpp.com/cpp-tutorial/object-slicing/#comment-579013
18. https://g.ezoic.net/privacy/learncpp.com