Dynamic Polymorphism with Metaclasses and Code Injection - Sy Brand - CppCon 2020

The polymorphism is the provision of a single interface to entities of different types.

|  |  |
| --- | --- |
| **Dynamic Polymorphism** | **Static Polymorphism** |
| Run-time | Compile-time |
| Different behavior based on dynamic type | Different behavior based on static type |
| Typically implemented with inheritance | Typically implemented with overloading and templates |
|  |  |
|  |  |
|  |  |

“Inheritance Is The Base Class of Evil” – Sean Parent (<https://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil>)

(обяснява много проблеми на наследяването)

**Problems with inheritance**

* Often requires dynamic allocation, where you might not needed;

Пример:

struct base{};

struct a : base{};

struct b : base{};

base make\_base();

std::vector<base> v;

Имаме функция, която връща base by value или вектор от base обекти.

Then this is going to do what’s calles “slicing” where we get just the base object a part of the object and we slice of the dynamic part of the derived object. Usually this is not what we want.

In order to make this work we have to do something like a unique\_ptr. We dynamically allocate we return a pointer, or we store a pointer in a vector

std::unique\_ptr<base> make\_base();

std::vector<std::unique\_ptr<base>> v;

And there are ways you could get around this by like building something on top of inheritance which does like a small buffer optimization or always uses in place storage, but by default inheritance and virtual functions you need to dynamically allocate if you want to use them in polymorphic way like this.

* Ownership and nullability considerations

Another problem is ownership and nullability. Since we have pointers, we have to care about the ownership. We have to care if this thing could be null. If we use unique\_ptr then the ownership is clear. But less clear is like if this thing is returning unique\_ptr. Can it return null. Do I have to check? If this thing is taking a unique\_ptr, what happens if I pass null? Is that valid? What’s the behavior? Something we have to care about which maybe we don’t want to.

* Intrusive: requires modifying child classes

Another is intrusiveness. Supporting inheritance requires modifying child classes.

An example of this: Intrusive Polymorphism

namespace mylib {

struct base {

virtual void do\_thing();

};

}

namespace otherlib {

struct x {

virtual void do\_thing();

};

}

We have a base class in mylib and then we have some other library which has an x class.

They both have do\_thing() method which returns void and they’re virtual. You cannot allocate an instance of otherlib x and take a pointer to it through a mylib base.

mylib::base\* b = new otherlib::x;

This will not work because otherlib x does not say it inherits from mylib base. This can be a problem. Maybe we can’t change otherlib x like if we could just decorate it and say ‘Oh, we inherit from base’. Maybe our problem solved but maybe we can’t we don’t own the code there is some other constraints which stop us from doing this. So polymorphism with inheritance is intrusive.

* No more value semantics

We have pointers again. If we want value semantics then we have to build something on top. If we have like a virtual clone function which will dynamically allocate a pointer and pass it back whish uses the correct dynamic type. And that’s a way of getting a copy behaviour but you still don’t have the usial C++ value semantics which a lot of code depends on.

* Changes semantics for algorithms and containers

And similarly it changes semantics for algorithms and containers like if you are doing a sort suddenly maybe we’re sorting on pointers and we have to suply our custom comparator object. Or if we are storing this things in a set, we might need to do the same thing. So it is another thing we are thinking about where we’re not in the usual world of C++ values which is where we like to be for most of our C++ development.

What we are going to do is implement virtual functions by hand. First, we are going to do it in normal C++ which you can write today and then we are going to improve it using techniques which might be coming in future versions. A lot of code which I’m showing is kind of complex and there is a bunch of new stuff and new features, new syntax you might not be familiar with.

struct animal {

virtual ~animal();

virtual void speak() = 0;

};

struct cat : animal {

void speak() override;

};

struct dog : animal {

void speak() override;

};

So this is a class hierarchy I’m going to try and emulate. I’ll try and implement by hand.

We have an animal class, which has a speak() function, which is pure virtual. And we have a virtual destructor. Then we have a cat class and dog class which override the speak() function and they inherit from animal. Fairly straightforward.

Now how this is implemented under the covers is that if I have a Felix object which is a cat then this is going to have a pointer to a vtable, a virtual table.

A picture containing diagram

Description automatically generated

This virtual table essentially tells us how to call functions of what virtual functions in a polymorphic object. So this vtable will in turn have a pointer to the speak() function for cat. So there is a couple of indirections we are going through here when we call speak(). First we have to grab the vtable, Then we have to read through the vtable to get the speak() function. That can be a performance bottleneck.

This is what I want:

struct animal {

// magic

};

struct cat {

void speak();

};

struct dog {

void speak();

};

This is the interface I want. I want to have an animal class which has a bunch of magic which I’m gonna implement and then I have a cat class and a dog class which both just have speak functions. Note that these are not inheriting from the animal class. But I want to support this uscase. I want to be able creant an animal from a cat and call speak(). And I want to be able to create an animal from a dog and I have it speak and not do any slicing.

int main() {

animal c = cat{};

c.speak();

animal d = dog{};

d.speak();

}

* Hand-written virtual functions

So now we’ll implement the magic in animal.

This is the steps we’re going to go through to implement our virtual functions by hand:

* Declare vtable for the abstract interface
* Define vtable for a concrete type
* Capture the vtable pointers on construction
* Forward calls through the vtable

First, we are going to declare what the vtable layout looks like for animal class. Then we are going to fill in the vtable for out cat and dog class. Then we will on construction of an animal. We are going to capture the vtable pointers for the corret concrete type. And the finally we call speak for the destructor we’re going to forward our calls through the vtable.

Започваме поред тези 4 стъпки:

* Declare vtable for the abstract interface

Our vtable is going to have 2 function pointers – one for speak and one for destroy which is going to call the destructor of the concrete object and also because we’re going to be allocating inside our object it’s going to reclaim that memory. These things are taking void pointers, because that’s how we’re going to store our concrete object internally. We’re going to pass void pointers around and then the concrete objects are going to cast those pointers internally.

struct vtable {

void (\*speak)(void\* ptr);

void (\*destroy\_)(void\* ptr);

};

* Define vtable for a concrete type

Now we’re going to define vtable for a concrete type.

template<class Concrete>

constexpr vtable vtable\_for{

// function which calls speak

// function which deletes object

};

This is a variable template. So we’re going to have an instance of a vtable and it’s going to be templated on the concrete type, i.e. cat or dog. So we need a function which is going to call the correct version of speak and a function which deletes the object and calls the destructor. And we can use lambdas for this.

template<class Concrete>

constexpr vtable vtable\_for{

[](void\* ptr) { static\_cast<Concrete\*>(ptr)->speak(); },

[](void\* ptr) { delete static\_cast<Concrete\*>(ptr); }

};

So, for a given concrete object the first function pointer is just going to static cast to the concrete type and then call speak(). And then the destroy() function is going to static cast and then called delete.

This all works because lambdas which don’t capture can decay to function pointers. Which is kind of nice.

* Capture the vtable pointers on construction

Next, we need to capture the vtable pointers. So when we construct an animal class what we need to do is fill in our pointer for our concrete object and pointer to our vtable.

struct animal {

void\* concrete\_;

vtable const\* vtable\_;

};

So our constructor is going to be a template. It’s going to take anything. Of course in the real world we would like constrain this but slideware and we’re gonna dynamically allocate a copy of what we’re given and store in concrete and then we’re gonna take a pointer to our vtable ans store that inside our animal.

struct animal {

void\* concrete\_;

vtable const\* vtable\_;

template<class T>

animal(T const& t) :

concrete\_(new T(t)),

vtable\_(&vtable\_for<T>)

{}

};

Now note that because we’ have access to what ‘t’ is at this point whether it’s a cat ot a dog. So this is where we’re kind of saving this information for later by grabbing the right vtable pointer and by dynamically allocating a copy of our ‘t’. This technique is called “type erasure” .

* Forward calls through the vtable

Finally we need to forward our calls through the vtable. Which looks like this:

struct animal {

// ...

void speak() { vtable\_->speak(t\_); }

~animal() { vtable\_->destroy\_(t\_); }

};

Inside our anumal class if we call speak() then we indirect through vtable and pass it our void pointer. And remember that is then going to cast inside the function and call the right version. And then similarly for the structure we call destroy().

So now we have something which works for that use case. We can have a struct cat and a dog and speak() functions. We can construct cat, we can construct an animal from a cat and a dog and make them speak and all it works.

struct cat {

void speak();

};

struct dog {

void speak();

};

int main() {

animal c = cat{};

c.speak();

animal d = dog{};

d.speak();

}

So let’s loook at what this has gained us. These are the problems we had with inheritance at the start. And we’ve solve some of these. We have solved the ownership and nullability because now all our memory allocations are handled inside the animal class. We don’t have pointers externally. We’re just dealing with the values. We’ve god rid of the intrusivity because remember cat and dog did not derive from anything. It just worked.

* Problems with inheritance

However we still have problems with “No move value semantics”, because we can’t copy these objects, bu we can solve that. We can extend our vtable with a clone() function and a move\_clone() function.

struct vtable {

void (\*speak)(void\* ptr);

void (\*destroy\_)(void\* ptr);

void\* (\*clone\_)(void\* ptr);

void\* (\*move\_clone\_)(void\* ptr);

};

And these do roughly what you’d expect – clone() will allocate a copy, move\_clone() will allocate by moving from the object.

template<class T>

constexpr vtable vtable\_for{

[](void\* ptr) { static\_cast<T\*>(ptr)->speak(); },

[](void\* ptr) { delete static\_cast<T\*>(ptr); },

[](void\* ptr) -> void\*

{ return new T(\*static\_cast<T\*>(ptr)); },

[](void\* ptr) -> void\*

{ return new T(std::move(\*static\_cast<T\*>(ptr))); }

};

Then in our copy constructor and move constructor for animal we just call from out vtable to clone and move\_clone and we make shure to copy over the vtable pointer from the object we’re created from. And then you would do the same thing for the copy assignment and move assignment operators.

struct animal {

//...

animal(animal const& rhs) :

t\_(rhs.vtable\_->clone\_(rhs.t\_)),

vtable\_(rhs.vtable\_)

{}

animal(animal&& rhs) :

t\_(rhs.vtable\_->move\_clone\_(rhs.t\_)),

vtable\_(rhs.vtable\_)

{}

};

int main() {

animal a = cat{};

a.speak();

a = dog{};

a.speak();

animal b = a;

b.speak();

}

Now we have actually kind of a nice interface. We can create animal from a cat, we cam make it speak, we can reassign it to a dog and meka that speak. We can create a new animal from an old one and make that speak. All works and we have value semantics even though we’re doing dynamic polymorphism. It’s just all handled under the covers.

You can even make a vector of these things like because we defined copying anm moving and things like that.

int main() {

std::vector<animal> animals{ cat{}, dog{} };

for (auto&& a : animals) {

a.speak();

}

}

We can put a cat and dog in a vector of animals and this will do what you expect.

Решихме още един проблем на наследяването. Вече имаме нормална copy семантика и container semantics. As long as we define any of the relevant like comparison operators and things like that for our animal class.

The dynamic allocation („Often requires dynamic allocation“) засега оставяме настрана.

Проблемът е, че написахме твърде много код, за да реализираме тази функционалност. Не искаме да правим това за всвки клас, който искаме да третираме динамично. И лесно да се обърка нещо, защото това е weird code.

(video 17:04) – Static Reflection