C++ corner case: You can implement pure virtual functions in the base class

devblogs.microsoft.com/oldnewthing/20131011-00

October 11, 2013

Raymond Chen

In our discussion `__purecall`, we saw that you can declare a pure virtual function with the `= 0` syntax, and if you try to call one of these functions from the base class, you will get the dreaded *R6025 – pure virtual function call* error.

In that article, I wrote that a pure virtual function is "a method which is declared by the base class, but for which no implementation is provided." That statement is false.

You can provide an implementation for a pure virtual method in C++.

"That's crazy talk," I hear you say.

Okay, let's start talking crazy:

```
#include <iostream>
class Base {
public:
 Base() { f(); }
 virtual void f() = 0;
};
class Derived : public Base {
public:
 Derived() { f(); }
 virtual void f() { std::cout << "D"; }
};
int main(int, char **)
{
 Derived d;
 return 0;
}
```

What happens when the `test` function constructs a `Derived`?

Trick question, because you get a linker error when you try to build the project.

There are many questions lurking here, like “Why do I get a linker error?” and “Why isn’t it a compiler error?” We’ll get back to those questions later. For now, let’s get the code to build.

```
class Base {
public:
 Base() { call_f(); }
 virtual void f() = 0;
 void call_f() { f(); }
};
```

Okay, with this change (hiding the call to `f` inside a function called `call_f`), the code compiles, so now we can answer the question:

Answer: You get the dreaded purecall error, because the base class constructor has engaged in a conspiracy with the function `call_f` to call the function `f` from its constructor. Since `f` is a pure virtual function, you get the purecall error.

Okay, next question: Why didn’t the original code result in a compiler error?

Answer: The compiler is not required to do a full code flow analysis to determine whether you are calling a pure virtual method from a constructor. The language forbids it, but no diagnostic is required and the behavior is undefined.

> Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.

But why did it result in a linker error?

Answer: As we learned during the discussion of `__purecall`, C++ objects change identity during their construction, and at the point that the `Base` constructor is running, the object is still a `Base`, so the *final overrider* method is `Base::f` Therefore, when you called `f()` from the `Base` constructor, you were actually calling `Base::f`.

And you never defined `Base::f`, so the linker complained.

“Wait, I can define `Base::f`?”

Sure, let’s do it. *At the end of the program* (even after the definition of `main`) add this code:

```
void Base::f()
{
 std::cout << "B";
}
```

Now the program compiles, and when you run it, well, we saw that the standard leaves this undefined, so you might crash, or monkeys might fly out of your nose, or the runtime library may go back in time and kill your parents. (We'll see in a future article how undefined behavior can lead to time travel. How's that for a teaser!)

But one implementation might generate this program output:

```
BD
```

This particular implementation decided not to try very hard to detect the case where you are calling `Base::f` during the constructor and just lets the call happen, and it ends up calling the method you defined later.

"But if I'm not allowed to call the pure virtual function from a constructor or destructor, and if I call the method after construction, it always calls the version of the function that the derived class overrode, then how could this code ever execute at all (legitimiately)? In other words, what conforming program could ever print the letter `B`?"

The function cannot be called implicitly, but it can be called explicitly:

```
class Base {
public:
 Base() { /* f(); */ }
 virtual void f() = 0;
};
void Base::f()
{
 std::cout << "B";
}
class Derived : public Base {
public:
 Derived() { f(); }
 virtual void f() { std::cout << "D"; Base::f(); }
};
int main(int, char **)
{
 Derived d;
 return 0;
}
```

First, we got rid of the illegal call to `f()` in the base class constructor (to keep our code legit). Next, we adjusted our override version of `f` so that it calls the base class method after doing some custom work.

This time, the program prints `DB`, and the code is perfectly legitimate this time. No undefined behavior, nothing up my sleeve.

What happened here?

The derived class constructor called the `f` method, which maps to `Derived::f`. That function prints the letter D, and then it calls the base class version `Base::f` explicitly. The base class version then prints the letter B.

This is actually nothing new; this is how overridden methods work in general. The only wrinkle here is that the base class method can be called only via explicit qualification; there is no way to call it implicitly.

This was a rather long-winded way of calling out a weird corner case in C++ that most people don't even realize exists: A pure virtual function can have an implementation.

Raymond Chen

Follow