What is __purecall?

devblogs.microsoft.com/oldnewthing/20040428-00

April 28, 2004

Raymond Chen

Both C++ and C# have the concept of virtual functions. These are functions which always invoke the most heavily derived implementation, even if called from a pointer to the base class. However, the two languages differ on the semantics of virtual functions during object construction and destruction.

C# objects exist as their final type before construction begins, whereas C++ objects change type during the construction process.

Here's an example:

```
class Base {
public:
  Base() { f(); }
  virtual void f() { cout << 1; }
  void g() { f(); }
};
class Derived : public Base {
public:
  Derived() { f(); }
  virtual void f() { cout << 2; }
};
```

When a `Derived` object is constructed, the object starts as a `Base`, then the `Base::Base` constructor is executed. Since the object is still a `Base`, the call to `f()` invokes `Base::f` and not `Derived::f`. After the `Base::Base` constructor completes, the object then becomes a `Derived` and the `Derived::Derived` constructor is run. This time, the call to `f()` invokes `Derived::f`.

In other words, constructing a `Derived` object prints "12".

Similar remarks apply to the destructor. The object is destructed in pieces, and a call to a virtual function invokes the function corresponding to the stage of destruction currently in progress.

This is why some coding guidelines recommend against calling virtual functions from a constructor or destructor. Depending on what stage of construction/destruction is taking place, the same call to `f()` can have different effects. For example, the function `Base::g()` above will call `Base::f` if called from the `Base::Base` constructor or destructor, but will call `Derived::f` if called after the object has been constructed and before it is destructed.

On the other hand, if this sample were written (with suitable syntactic changes) in C#, the output would be "22" because a C# object is created as its final type. Both calls to `f()` invoke `Derived::f`, since the object is always a `Derived`. Notice that means **a method can be invoked on an object before its constructor has run**. Something to bear in mind.

Sometimes your C++ program may crash with the error "R6025 – pure virtual function call". This message comes from a function called __purecall. What does it mean?

C++ and C# both have the concept of a "pure virtual function" (which C# calls "abstract"). This is a method which is declared by the base class, but for which no implementation is provided. In C++ the syntax for this is "=0":

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

If you attempt to create a `Derived` object, the base class will attempt to call `Base::f`, which does not exist since it is a pure virtual function. When this happens, the "pure virtual function call" error is raised and the program is terminated.

Of course, the mistake is rarely as obvious as this. Typically, the call to the pure virtual function occurs deep inside the call stack of the constructor.

This raises the side issue of the "novtable" optimization. As we noted above, the identity of the object changes during construction. This change of identity is performed by swapping the vtables around during construction. If you have a base class that is never instantiated directly but always via a derived class, and **if you have followed the rules against calling virtual methods during construction**, then you can use the novtable optimization to get rid of the vtable swapping during construction of the base class.

If you use this optimization, then **calling virtual methods during the base class's constructor or destructor will result in undefined behavior**. It's a nice optimization, but it's your own responsibility to make sure you conform to its requirements.

Sidebar: Why does C# not do type morphing during construction? One reason is that it would result in the possibility, given two objects A and B, that typeof(A) == typeof(B) yet sizeof(A) != sizeof(B). This would happen if A were a fully constructed object and B were a partially-constructed object on its way to becoming a derived object.

Why is this so bad? Because the garbage collector is really keen on knowing the size of each object so it can know how much memory to free. It does this by checking the object's type. If an object's type did not completely determine its size, this would result in the garbage collector having to do extra work to figure out exactly how big the object is, which means extra code in the constructor and destructor, as well as space in the object, to keep track of which stage of construction/destruction is currently in progress. And all this for something most coding guidelines recommend against anyway.

Raymond Chen

Follow