C++/WinRT injects additional constructors into each runtime class

devblogs.microsoft.com/oldnewthing/20200907-00

September 7, 2020

Raymond Chen

C++/WinRT treats runtime classes similar to C# reference types. Copying a C++/WinRT runtime class copies a reference to the underlying object. You can null out the reference by assigning `nullptr` to it.

On the other hand, C++ constructors don't use the `new` keyword; the `new` keyword has a different meaning which doesn't apply to Windows Runtime classes.

This means that C++ constructors have to do double-duty: They can be used to construct new objects, or they can be used as copy constructors or conversion constructors.

Constructors that actually, y'know, create new objects are represented as traditional C++ constructors.

C++/WinRT also injects additional constructors into each runtime class. One is the copy constructor, and another is the conversion constructor from `nullptr`.

If you had a class that has a default constructor, or could construct from an integer, you would write it something like this:

```
class Thing
{
public:
    Thing();
    explicit Thing(int capacity);
};
```

The C++/WinRT version looks similar, but with additional constructors:

```
class Thing
{
public:
    Thing();
    explicit Thing(int capacity);
    Thing(std::nullptr_t);
    Thing(Thing const&) = default;
    Thing(const&&) = default;
};
```

(If you look at the C++/WinRT headers, you won't see the default constructors. They simply are generated automatically by the compiler.)

The first injected constructor is the conversion constructor from `nullptr`. The second and third are the copy and move copy constructors, which copy or move the reference to the underlying object.

```
// default constructor, creates an object
Thing t1;

// explicit constructor, creates an object
Thing t2{ 42 };

// conversion from nullptr, creates an empty reference
Thing t3{ nullptr };
Thing t4 = nullptr;

// copy constructor, copies reference to object
Thing t5{ t1 };
Thing t6 = t1;

// move copy constructor, moves reference to object
Thing t5{ std::move(t1) };
Thing t6 = std::move(t1);
```

This conflation of reference construction and object construction can be confusing. For example, you might forget that the default constructor creates an object:

```
class Something
{
private:
    Thing m_thing;
};
```

This constructs a brand new `Thing` object when the `Something` constructs. If you wanted to start with an empty reference, you need to initialize `m_thing` with `nullptr`.

```
class Something
{
private:
    Thing m_thing = nullptr;
};
```

When designing your own runtime classes, you may want to avoid having a constructor whose single parameter is the same as the type being constructed, because that would conflict with the copy constructor.

```
runtimeclass Thing
{
  Thing(Thing parent);
}
```

This would result in two conflicting projections into C++/WinRT. Would

```
// assuming t1 is a Thing
Thing t2{ t1 };
```

be an attempt to construct a brand new `Thing`, using `t1` as the constructor parameter? Or would it be an attempt to copy the reference to the same underlying `Thing` object?

You can work around this by using a static function that acts like a constructor.

```
runtimeclass Thing
{
  static Thing CreateFromParent(Thing parent);
}
```

Or by changing it to a method on the parent.

```
runtimeclass Thing
{
  Thing CreateChild();
}
```

There is also an ambiguity if you have a constructor that takes a single reference type, and it’s possible for that reference to be null.

```
runtimeclass ChildThing
{
  // parent=null means create parentless
  ChildThing(ParentThing parent);
}
```

In this case, you might be tempted to create a new parentless `ChildThing` object by saying this:

```
ChildThing child{ nullptr };
```

Unfortunately, this actually invokes the conversion constructor from `nullptr`. To construct a new parentless `ChildThing`, you need to write

```
ChildThing child{ ParentThing{ nullptr } };
```

A cleaner workaround is to provide a default constructor that creates a parentless `ChildThing`.

```
runtimeclass ChildThing
{
  ChildThing(); // create parentless
  ChildThing(ParentThing parent); // creates with parent
}
```

Raymond Chen

Follow