A static_cast is not always just a pointer adjustment

devblogs.microsoft.com/oldnewthing/20160224-00

February 24, 2016

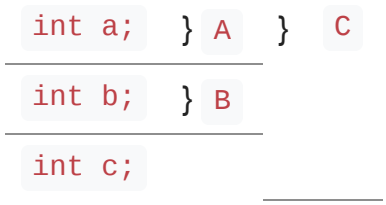
Raymond Chen

Even without considering virtual base classes, a `static_cast` to move between a base class and a derived class can be more than just a pointer adjustment.

Consider the following classes and functions.

```
class A
{
public:
 int a;
 void DoSomethingA();
};

class B
{
public:
 int b;
 void DoSomethingB();
};

class C : public A, public B
{
public:
 int c;
 void DoSomethingC();
};

B* GetB(C* c)
{
 return static_cast<B*>(c);
}

void AcceptB(B* b);

void AcceptC(C* c)
{
 AcceptB(c);
}
```

Suppose the compiler decided to lay out the memory for `C` like this:



Now, you would think that converting a pointer to a `C` into a pointer to a `B` would be a simple matter of adding `sizeof(int)`, since that's what you need to do to get from the `a` to the `b`.

Unless you happen to have started with a null pointer.

The rule for null pointers is that casting a null pointer to anything results in another null pointer.

This means that if the parameter to `GetB` is a null pointer, the function cannot return `nullptr + sizeof(int)`; it has to return `nullptr`.

```
GetB:
    xor rax, rax
    test rcx, rcx
    jz @F
    lea rax, [rcx+sizeof(int)]
@@: ret
```

Similarly, if the parameter to `AcceptC` is `nullptr`, then it must call `AcceptB` with `nullptr`.

```
AcceptC:
    test rcx, rcx
    jz   @F
    add  rcx, sizeof(int)
@@: jmp  AcceptB
```

A naïve compiler would insert all these conditional jumps every time you cast between a base class and a derived class that involves an adjustment. But this is also a case where a compiler that takes advantage of undefined behavior can optimize the test away: If it sees that every code path through the `static_cast` dereferences either the upcast or downcast pointer, then that means that if the pointer being converted were `nullptr`, it would result in undefined behavior. Therefore, the compiler can assume that the pointer is never `nullptr` and remove the test.

```
void AcceptC2(C* c)
{
 c->DoSomethingB();
}
```

Here, the test can be elided because the result of the conversion is immediate dereferenced in order to call the `B::DoSomethingB` method. The C++ language says that if you try to call an instance method on a null pointer, the behavior is undefined. Doesn't matter whether the method actually accesses any member variables; just the fact that you invoked an instance method is enough to guarantee that the pointer is not null. Therefore, the `AcceptC2` function compiles to

```
AcceptC2:
    add rcx, sizeof(int)
    jmp B::DoSomethingB
```

The same logic applies on the receiving end of the method call: A method call can assume that `this` is never null.

```
void C::DoSomethingC()
{
 AcceptB(this);
}

C::DoSomethingC:
    add rcx, sizeof(int)
    jmp AcceptB
```

Since `this` is never null, the conversion from `C*` to `B*` can elide the test and perform the adjustment unconditionally.

This means that you could add a dummy method to ever class:

```
class C : public A, public B
{
public:
 void IsNotNull() { }
 int c;
 void DoSomethingC();
};
```

and call `c->IsNotNull()` to tell the compiler, "I guarantee on penalty of undefined behavior that `c` is not null."

```
void AcceptC3(C* c)
{
 c->IsNotNull();
 AcceptB(c);
}

AcceptC3:
    add rcx, sizeof(int)
    jmp AcceptB
```

I don't know whether any compilers actually take advantage of this hint, but at least this is a way of providing it in a standard-conforming way.

Now, it looks like the purpose of this article is to delve into optimization tweaking in order to remove unwanted tests, but that wasn't actually the point. The point of the article was to explain what these tests are for. You'll be stepping through some code, and you'll see these strange tests against zero, so here's an explanation of why those tests are there.

Raymond Chen

Follow