Adventures in undefined behavior: The premature downcast

devblogs.microsoft.com/oldnewthing/20120831-00

August 31, 2012

Raymond Chen

A customer encountered the following problem:

```
class Shape
{
public:
    virtual bool Is2D() { return false; }
};
class Shape2D : public Shape
{
public:
    virtual bool Is2D() { return true; }
};
Shape *FindShape(Cookie cookie);
void BuyPaint(Cookie cookie)
{
    Shape2D *shape = static_cast<Shape2D *>(FindShape(cookie));
    if (shape->Is2D()) {
       .. do all sorts of stuff ...
    }
}
```

The `BuyPaint` function converts the cookie back to a `Shape` object, and then checks if the object is a `Shape2D` object by calling `Is2D`. If so, then it does some more stuff to figure out what type of paint to buy.

(Note to nitpickers: The actual scenario was not like this, but I presented it this way to illustrate the point. If you say "You should've used RTTI" or "You should've had a BuyPaint method on the Shape class", then you're missing the point.)

The programmers figured they'd save some typing by casting the result of `FindShape` to a `Shape2D` right away, because after all, since `Is2D` is a virtual method, it will call the right version of the function, either `Shape::Is2D` or `Shape2D::Is2D`, depending on the actual type of the underlying object.

But when compiler optimizations were turned on, they discovered that the call to `Is2D` was optimized away, and the `BuyPaint` function merely assumed that it was always operating on a `Shape2D` object. It then ended up trying to buy paint even for one-dimensional objects like points and lines.

Compiler optimization bug? Nope. Code bug due to reliance on undefined behavior.

The C++ language says (9.3.1) "If a nonstatic member function of a class `X` is called for an object that is not of type `X`, or of a type derived from `X`, the behavior is undefined." In other words, if you are invoking a method on an object of type `X`, then you are promising that it really is of type `X`, or a class derived from it.

The code above violates this rule, because it is invoking the `Is2D` method on a `Shape2D*`, which therefore comes with the promise "This really is a `Shape2D` object (or something derived from it)." But this is a promise the code cannot deliver on, because the object returned by `FindShape` might be a simple `Shape`.

The compiler ran with the (false) promise and said, "Well, since you are guaranteeing that the object is at least a `Shape2D`, and since I have studied your code and determined that no classes which further derive from `Shape2D` override the `Is2D` method, I have therefore proved that the *final overrider* is `Shape2D::Is2D` and can therefore inline that method."

Result: The compiler inlines `Shape2D::Is2D`, which returns `true`, so the "if" test can be optimized out. The compiler can assume that `BuyPaint` is always called with cookies that represent two-dimensional objects.

The fix is to do the annoying typing that the original authors were trying to avoid:

```
void BuyPaint(Cookie cookie)
{
    Shape *shape = FindShape(cookie);
    if (shape->Is2D()) {
      Shape2D *shape2d = static_cast<Shape2D *>(shape);
       .. do all sorts of stuff (with shape2d) ...
    }
}
```

Raymond Chen

Follow