Exposing undefined behavior when trying to port code to another platform

devblogs.microsoft.com/oldnewthing/20171222-00

December 22, 2017

Raymond Chen

A developer was porting some old Visual Studio code to another platform and found that the code was behavior strangely. Here's a simplified version of the code:

```
class refarray
{
public:
    refarray(int length)
    {
        m_array = new int*[length];
        for (int i = 0; i < length; i++) {
            m_array[i] = NULL;
        }
    }

    int& operator[](int i)
    {
        return *m_array[i];
    }

    ... other members not relevant here...

private:
    int** m_array;
};
```

This class is an array of references to integers. Each slot starts out uninitialized, but you can use methods (not shown here) to make each slot point to a particular integer, and you use the array indexing operator to access the referenced integer. (You can tell this is old code because it's not using `unique_ptr` or `reference_wrapper` or `nullptr`.)

Here's some typical code that didn't work:

```
refarray frameCounts(NUM_RENDERERS);

void refresh(int* frameCount)
{
    .. a bunch of refresh code ..
    if (frameCount != NULL) ++*frameCount;
}

void refresh_and_count(int i)
{
    refresh(&frameCounts[i]);
}
```

The `refresh` function performs a refresh and if the pointer is non-null, it assumes it's a frame count and increments it. The `refresh_and_count` function uses the `refresh` function to perform an update and then increment the optional frame counter stored in the `frameCounts` object.

The developer found that if the slot was not set, the code crashed with a null pointer access violation at the `++*frameCount`, despite the code explicitly checking `if (frameCount != NULL)` immediately prior.

Further investigation showed that the code worked fine with optimization disabled, but once they started turning on optimizations, the null pointer check stopped working.

The developer fell into the trap of the null reference, or more generally, the fact that undefined behavior can have strange effects.

In the C++ language, there is no such thing as a null reference. All references are to valid objects. The expression `frameCounts[i]` produces a reference, and therefore the expression `&frameCounts[i]` can never legally produce a null pointer. The compiler optimized out the null test because it could prove that the resulting pointer could never legally be null.

The code worked on the very old of Visual Studio because very old Visual Studio compilers did not implement this optimization. They generated the pointer and redundantly tested it against null, even though the only way to generate such a null pointer was to break one of the rules of the language.

The new compiler on that other platform took advantage of the optimization: After one level of inlining, the compiler noticed that the pointer could not be null, so it removed the test.

The fix is to repair the code so it doesn't generate null references.

I know that people will complain that the compiler should not be removing reundant tests, because the person who wrote the code presumably wrote the redundant tests for a reason. Or at least if the compiler removed the redundant test, it should emit a warning: "Removing

provably false test."

But on the other hand, surely you would want the compiler to optimize out the test when you call it like this:

```
int counter;
void something()
{
    refresh(&counter);
}
```

This is another case where the pointer passed to `refresh` is provably non-null. Do you want the compiler to generate the test anyway? If not, then it would presumably generate the "Removing provably false test" warning. Your code would probably generate tons of instances of this warning, and none of your options look appealing.

One option is to duplicate the `refresh` function into one version that supports a null pointer (and performs the test), and another version that requires a non-null pointer (and doesn't perform the test). This sort of change can quickly infect your entire code, because callers of `refresh` might in turn need to split into two versions, and pretty soon you have two versions of half of your program.

The other option is to suppress the warning.

In practice, you're probably going to go for the second option.

But there's clearly no point in the compiler team implementing a warning that everybody suppresses.

Raymond Chen

Follow