Why am I being told about a signed/unsigned comparison, and why only sometimes, and how can I fix it?

devblogs.microsoft.com/oldnewthing/20230619-00

June 19, 2023

Raymond Chen

A customer was using the Windows Test Authoring and Execution Framework (TAEF), and they found that this line of code compiled successfully most of the time:

```
std::vector<int> values = CalculateValues();
VERIFY_ARE_EQUAL(values.size(), 0);
// warning C4389: '==': signed/unsigned mismatch
```

The `VERIFY_ARE_EQUAL` macro compares its two parameters and reports a test failure if they are not equal. The customer found that the above code compiled okay for x86-64, but produced the indicated error when compiled for x86-32. What's going on?

The `VERIFY_ARE_EQUAL` macro passes its parameters onward to a helper function, which in turn passes the parameters to another helper function, which in turn passes the parameters to yet another helper function, which looks like this:

```
template<typename T1, typename T2>
static bool AreEqual(const T1& expected, const T2& actual)
{
    // != 0 to handle overloads of operator==
    // that return BOOL instead of bool
    return (expected == actual) != 0;
}
```

The types of the parameters are deduced as `std::vector<int>::size_type` and `int`. The first parameter is a `std::vector<int>::size_type`, because that's what the `vector::size()` method returns. The second parameter is a `int` because that's what `0` is.

Therefore, the compiler warning of an unsigned/signed comparison is valid. You are comparing an unsigned value (the size of the vector) against a signed value (the integer zero). The warning is present because the rules for an unsigned/signed comparison is to convert the signed value to unsigned, and then compare the two unsigned values. This is different from the mathematical result.

```
unsigned int v1 = 4294967295;
int v2 = -1;
if (v1 == v2) { /* true */ }
if (v1 > v2) { /* false */ }
```

Both results disagree with the mathematical expectations. 4294967295 ≠ −1, so mathematically, the first test should fail. And 4294967295 > −1, so mathematically, the second test should succeed. What's happening is that the value −1 is converted from a signed integer to unsigned, and that means that it becomes 4294967295. (The rule for signed-to-unsigned conversion is that negative numbers become the positive equivalent modulo `1 << bit_size`.)

You and I can see that this discrepancy doesn't apply here because the signed integer being compared against is the literal value zero, which is not negative.

What's happening is that the back-end applies different degrees of inlining based on optimization levels and the target architecture. Higher optimization levels will consider higher degrees of inlining, and some architectures may be more conducive to inlining than others, depending on things like register pressure, the complexity of the calling convention, or other factors.

If the back-end ends up doing enough inlining that the constant `0` gets inlined into the `==` operator, then the compiler realizes, "Oh, I would normally complain about a signed/unsigned mismatch, but I can see that the value `0` is not negative, so I will suppress the warning because it doesn't apply to this case." On the other hand, if the back-end doesn't inline aggressively enough, it won't propagate the constant deep enough to realize that it's never negative, and you get the warning.

The customer offered this solution but complained that it was quite unwieldy:

```
VERIFY_ARE_EQUAL(values.size(), (std::vector<int>::size_type)0);
```

That will work, but so too will casting to any other unsigned type, because the compiler is not insisting that the second parameter's type match the first parameter's type exactly. It just wants them to agree on signedness. You could pass a `size_t` or even a `uint8_t`; as long as it's unsigned. And probably the most convenient way to indicate an unsigned zero is to append a `U` to the literal.

```
VERIFY_ARE_EQUAL(values.size(), 0U);
```