Inadvertently passing large objects by value

devblogs.microsoft.com/oldnewthing/20060329-02

March 29, 2006

Raymond Chen

One mark of punctuation can make all the difference.

One program was encountering a stack overflow exception in a function that didn't appear to be doing anything particularly stack-hungry. The following code illustrates the problem:

```
bool TestResults::IsEqual(TestResults& expected)
{
 if (m_testMask != expected.m_testMask) {
  return false;
 }
 bool result = true;
 if (result && (m_testMask & AbcTestType)) {
  result = CompareAbc(expected);
 }
 if (result && (m_testMask & DefTestType)) {
  result = CompareDef(expected);
 }
 if (result && (m_testMask & GhiTestType)) {
  result = CompareGhi(expected);
 }
 if (result && (m_testMask & JklTestType)) {
  result = CompareJkl(expected);
 }
 return result;
}
```

(In reality, the algorithm for comparing two tests results was much more complicated, but that's irrelevant to this discussion.)

And yet on entry to this function, a stack overflow was raised.

The first thing to note is that this problem occurred only on the x64 build of the test. The x86 version ran fine, or at least appeared to. It so happens that the x64 compiler aggressively inlines functions, which as it turned out was a major exacerbator of the problem.

The title of this entry probably tipped you off to what happened: The helper functions accepted the test results parameter by value not by reference:

```
bool TestResults::CompareAbc(TestResults expected);
bool TestResults::CompareDef(TestResults expected);
bool TestResults::CompareGhi(TestResults expected);
bool TestResults::CompareJkl(TestResults expected);
```

and those comparison functions in turn called other comparison functions, which also passed the `TestResults` by value. Since the test results were passed by value, a temporary copy was made on the stack and passed to the comparison function. It so happened that the `TestResults` class was a very large one, a hundred kilobytes or so, and the `TestResults::IsEqual` function therefore needed to reserve room for a large number of such temporary copies, one for each call to a comparison function in each of the inlined functions. A dozen temporary copies times a hundred kilobytes per copy comes out to over a megabyte of temporary variables, which exceeded the default one megabyte stack size and therefore resulted in a stack overflow exception on entry to the `TestResults::IsEqual` function.

This code appeared to run fine when compiled for the x86 architecture because the x86-targetting compiler did not inline quite as aggressively, so the large temporaries were not reserved on the stack until the helper comparison was actually called. Since the comparisons went only three levels deep, there were only three temporary copies of the `expected` parameter, which fit within the one megabyte default stack. It was still bad code—consuming a few hundred kilobytes of stack for no reason—but it wasn’t bad enough to cause a problem. The fix, of course, was to change the comparison functions to accept the parameter by reference.

```
bool TestResults::IsEqual(const TestResults& expected) const;
bool TestResults::CompareAbc(const TestResults& expected) const;
bool TestResults::CompareDef(const TestResults& expected) const;
bool TestResults::CompareGhi(const TestResults& expected) const;
bool TestResults::CompareJkl(const TestResults& expected) const;
```

For good measure, the parameter was changed to a `const` reference, and the function was tagged as itself `const` to emphasize that neither the object nor the expected value will be modified as part of the comparison, thereby ensuring that changing from a copy to a `const` reference didn’t change the previous behavior. Without the `const` reference, there was a possibility that somewhere deep inside the comparison functions, they made a change to the `expected` parameter. Under the old pass-by-value declaration, this change was discarded when the function returned since the change was made to a copy. If we had left off the `const` from the reference, then we would have changed the behavior: The change to the `expected` parameter would have modified the original `TestResults`. Making the parameter `const` reassures us that an attempt to modify `expected` would be flagged by the compiler and therefore brought to our attention.

(This technique is not foolproof, however. Somebody could always cast away `const` -ness and modify the original, but we were being reckless and assuming that nobody would be that crazy.)

Raymond Chen

Follow