Is it true that raising a structured exception from a structured exception handler terminates the process?

devblogs.microsoft.com/oldnewthing/20221020-00

October 20, 2022

Raymond Chen

A customer had a vague recollection that they had read somewhere that if you raise a structured exception from a structured exception handler, the operating system would terminate the process. However, they couldn't find any confirmation of this behavior. Was it just a dream?

When you write a Windows structured exception handler (which is different from a C++ exception handler), you provide two code fragments:

- The code to decide whether to handle the exception.
- The code to execute if the exception is handled.

Let's annotate some code that handles a structured exception:

```
__try
{
    Block1;
}
__except
(FilterExpression)
{
    Block2;
}
__finally
{
    Block3;
}
```

Now, you aren't allowed to have both an `__except` block and a `__finally` block, so the above code is technically incorrect, but I'm going to use it because it lets me talk about all the parts of the structured exception handler in a single (if impossible) example. In reality, either the `__except` block or the `__finally` block will be missing, in which case you can just ignore that part of the discussion.

If a structured exception is raised in the `__try` block labeled `Block1`, then the `__except` block's filter expression is evaluated. If the filter returns `EXCEPTION_EXECUTE_HANDLER`, then the exception is considered to have been handled, and execution resumes at the `__except` block labeled `Block2`.

If a structured exception is raised in the `__except` block labeled `Block2` or in the `__finally` block labeled `Block3`, that structured exception is not considered to be in scope of this structured exception handler. The search for a handler begins at the next outer scope.

There is no automatic termination if a structured exception occurs in the `__except` block `Block2` or the `__finally` block `Block3`. The search for a handler proceeds in the usual fashion, but with the understanding that the exception is not protected by the `__try` statement. The search begins with the scope that *contains* the `__try` statement.

But wait, the story isn't over yet. There's still a place where an exception can be raised that I haven't talked about yet. Do you see it?

What if an exception is raised by the evaluation of the `FilterExpression`?

The filter expression is considered to be inside the scope of the `__try`, so the exception raised by the filter expression will cause a new evaluation of the filter expresion, but this time to evaluate the recursively raised exception.

That's the part that usually causes trouble.

If the evaluation of the filter expression for the first exception raises an exception, there's a good chance that evaluation of the filter expression for the nested exception will raise the same exception, because the nested exception is probably an access violation due to some bug in the filter expression.

You now run into a recursive exception death: To decide what to do about the original exception, the system evaluates the filter expression. But the filter expression has a bug, and it raises an exception. Now, the system decides what to do about the nested exception, which evaluates a new filter expression. That second filter expression encounters the same bug, so it raises a second nested exception. Each nested exception triggers a re-evaluation of the filter expression, and (on the assumption that the filter expression has a crashing bug) each re-evaluation in turn raises another nested exception.

Eventually, you run out of stack, and the unhandled stack overflow exception is what terminates the process.

Here's an annotated version of the above impossible example:

Under consideration

```
__try
{
    Block1;
}
__except
(FilterExpression)
```

Not considered

```
{
    Block2;
}
__finally
{
    Block3;
}
```

If an exception occurs in the code marked “under consideration”, then the filter expression participates in the handling of the exception. But if an exception occurs in the code marked “not considered”, then the filter expression does not participate; execution has left the exception scope of the `__try` statement.

Next time, we’ll look at the C++ version of this same question. The answer isn’t the same!

Bonus chatter: But wait, suppose we are using the `__try` / `__finally` version of this statement. If an exception is raised inside the `__try` block, and nothing in `Block1` handles the exception, then the `__finally` block will run. But what if the `__finally` block also raises an exception?

The system looks for a handler for the nested exception, and if an outer handler decides to handle it, then that handler executes, and the original exception is lost. On the other hand, if no filter expression declares that it wants to handle the exception, then the unhandled exception filter is called, and that terminates the process.

Either way, the original exception doesn’t get observed by any of the exceptions handlers that are in scope at the time the `__finally` block runs, so you can think of it as saying that the exception raised by the `__finally` block replaces the original exception.

Raymond Chen

Follow