If you want to terminate on an unexpected exception, then don’t sniff at every exception; just let the process terminate

devblogs.microsoft.com/oldnewthing/20191024-00

October 24, 2019

Raymond Chen

You’ve probably had to write code that lives at the boundary between exception-based code and error-code-based code.

```
HRESULT ConvertExceptionToHResult()
{
    try
    {
        throw;
    }
    catch (MyCustomExceptionClass const&amp ex)
    {
        return ex.GetHResult();
    }
    catch (std::bad_alloc const&amp)
    {
        return E_OUTOFMEMORY;
    }
    catch (...)
    {
        // Disallowed exception. Fail fast and get a crash dump.
        std::terminate();
    }
}

HRESULT DoSomething()
{
    try
    {
       DoStuffThatMayThrowExceptions();
       AnotherThingThatMayThrowExceptions();
    }
    catch (...)
    {
        return ConvertExceptionToHResult();
    }
}
```

The idea is that the `DoSomething` function uses an error code to report problems, but it is built with the help of functions that use exceptions to report problems. The `DoSomething` function sets up a `try` / `catch` that catches any exceptions that may emerge from the helper functions and uses a helper function to convert the exception to an `HRESULT`. If the exception cannot be converted to an `HRESULT`, then we terminate the process, because the helper functions threw a disallowed exception.

This works, but it does have a problem: When the inevitable crash reports arrive that say “Oh no, somebody threw a disallowed exception.” *The stack trace won’t tell you.*

Let’s illustrate with a quick little program.

```
#include <cstdlib>
#include <new>
#include <exception>
#include <errno.h>

struct MyCustomExceptionClass
{
    int code;
};

int oopsie()
{
    int value = std::rand();
    if (value >= 0) throw 1; // totally disallowed exception
    return value;
}

int victim() try
{
    return oopsie();
}
catch (MyCustomExceptionClass const& ex)
{
    return ex.code;
}
catch (std::bad_alloc const& ex)
{
    return ENOMEM;
}
catch (...)
{
   std::terminate();
}

int main()
{
    return victim();
}
```

I'm taking advantage of a feature known as the function try block that lets you float the `try` / `catch` outside the function body. This is handy because it saves you a level of indentation and makes it clearer (to those who have been initiated into the practice) that the `try` / `catch` block applies to the entire function body.

When you run this program, it crashes, and all you see on the stack is the `victim`.

```
_exit+0x11
abort+0xe8
terminate+0x3b
victim+0x5b ⇐ no sign of oopsie
main+0xd
```

You may be able to extract the object that was thrown, but the code that threw it has already left the building.

Why is that?

The problem is that your `catch (...)` was a *successful catch*. You said, “Sure, I’ll catch anything!” If the thrown object doesn’t match any of the earlier clauses, the runtime says, “Okay, the code says it’ll catch anything. That’s great! Let me do my stack unwinding, then. Destructing automatic variables whose scopes have exited. All that great RAII stuff.” After the destructors have run and the stack has unwound, execution resumes in your handler. You successfully handled the exception.

Of course, if your code that successfully handles the exception chooses to terminate the process, well, that’s your choice. But the code that threw the original exception is long gone.

The solution is simple: Don’t catch what you can’t handle.

```
void victim() noexcept try
{
    oopsie();
}
catch (MyCustomExceptionClass const& ex)
{
    return ex.code;
}
catch (std::bad_alloc const& ex)
{
    return ENOMEM;
}
// catch (...)
// {
//    std::terminate();
// }
```

We removed the `catch (...)` so that any exceptions we don’t understand are not handled. And then we added `noexcept` to the function signature to indicate that the process should terminate if an exception goes unhandled.

This time, the stack in the crash dump is more useful:

```
_exit+0x11
abort+0xe8
terminate+0x3b
FindHandler+0x377
__InternalCxxFrameHandler+0xf7
__CxxFrameHandler2+0x26
ExecuteHandler2+0x26
ExecuteHandler+0x24
KiUserExceptionDispatcher+0x26
RaiseException+0x62
_CxxThrowException+0x68
oopsie+0x2c ⇐ here's the bad boy
victim+0x3a
main+0x33
```

The C++ standard leaves it up to the implementation whether stack unwinding occurs, but the Visual C++ compiler does not unwind the stack. This means that the code that threw the exception is plain to see on the stack, and you can walk up the stack and look at local variables. Even better: No destructors have run, so the state of the process in the dump is the state at the time of the throw.

```
0:000> .frame 5
05 02a9fba0 00751e0a scratch!oopsie+0x2c
0:000> dv
          value = 0n41
```

Those local variables may end up being crucial to understanding why the disallowed exception was thrown.

C++/WinRT made this change in PR 423.

Raymond Chen

Follow