When I install an unhandled structured exception filter, why doesn't std::terminate get called?

devblogs.microsoft.com/oldnewthing/20250709-00/?p=111360

July 9, 2025

For diagnostic and reliability purposes, a customer wanted to detect and report all all unhandled C++ exceptions as well as all unhandled structured exceptions. Their idea was to cover both exits:

- Use `set_terminate` to install a `std::terminate` handler, which is called when a C++ exception goes unhandled.
- Use `SetUnhandledExceptionFilter` to install an unhandled exception handler, which is called when a structured exception goes unhandled.

When they did this, they found that C++ exceptions never reached their `std::terminate` handler. They instead went to the unhandled structured exception filter.

The customer understood that C++ exceptions are internally implemented by the Microsoft Visual C++ compiler in terms of structured exceptions but their understanding was that the unhandled structured exception filter is called last, after all other exception filters. So the C++ unhandled exception filter should have run first.

While it's true that the unhandled structured exception filter is called last, what the customer didn't realize is that the way that the Microsoft Visual C++ compiler recognizes unhandled exceptions is by itself installing an unhandled exception filter!

To throw a C++ exception, the Microsoft Visual C++ compiler calls `RaiseException`, and the operating system then walks up the stacks looking for any functions that have installed structured exception filters. The C++ compiler generates these structured exception filters for any function that requires awareness of exceptions, even if they don't themselves catch exceptions. For example, if there are local variables with destructors, the structured exception filter will destruct those local variables when an exception escapes their frame. (This is, after all, the principle behind RAII.)

If no functions on the stack handle the structured exception (which itself represents the C++ exception), then the custom unhandled structured exception filter installed by the Microsoft Visual C++ compiler inspects the exception, realizes that it's a C++ exception,

and concludes that what it has is an unhandled C++ exception.

Therefore, if you install your own custom unhandled structured exception filter, it will replace the unhandled structured exception filter installed by the Microsoft Visual C++ compiler, and therefore you will see the unhandled C++ exception instead of the compiler infrastructure.

Now that we understand what is happening, we can look for solutions next time.

Bonus chatter: If you think about it, the Visual C++ runtime doesn't have much choice but to install an unhandled structured exception filter. If a C++ exception goes unhandled, that means that the corresponding structured exception goes unhandled, and how else are you going to realize that this has happened aside from installing your own unhandled structured exception filter?

Okay, so one way to solve this would be to have the C++ runtime raw entry point install an exception handler filter. Since it is the outermost exception handler filter on the thread, it will get called for anything that escaped `main` unhandled. Similarly, the threads created by `_beginthreadex` and `std::thread` could install their own exception handler filter before calling the app-supplied thread function. However, this fails to observe unhandled exceptions that escape threads that were created by calling `CreateThread` directly, so you would have to add another rule that says "Threads created by `CreateThread` must not allow exceptions to escape." Since functions generally do not know how the thread they are running on was created, this rule breaks down quickly.