Detecting and reporting all unhandled C++ exceptions as well as all unhandled structured exceptions

devblogs.microsoft.com/oldnewthing/20250711-00/?p=111368

July 11, 2025

Last time, attempted to intercept the ways that a C++ program could exit due to an unhandled exception by installing a custom unhandled structured exception filter and letting it classify the nature of the unhandled exception. But I also noted that there are other ways a C++ program can exit on an unhandled exception that aren't covered by this. How can that be?

If you mark a function as `noexcept`, then the compiler will terminate execution if an unhandled exception exits that function. In that case, the unhandled exception never reaches the custom filter because the Microsoft Visual C++ runtime never lets the exception get that far.

Marking a function as `noexcept` is similar to manually wrapping the function body in a `try`/`catch` that "handles" the exception by terminating the program.[1]

```
void MyFunction() try
{
    ⟦ existing body ⟧
}
catch (...)
{
    std::terminate();
}
```

From the operating system's point of view, this exception was handled, which means that the unhandled exception filter is never called. Mind you, the exception was "handled" by explicitly terminating the program, but the operating system doesn't know or care about what the handler does.

Another source of unhandled exceptions is in coroutines. When an exception happens in a coroutine, the coroutine promise's `unhandled_exception()` is called, and each promise can decide how it wants to treat unhandled exceptions. One possible response is to call `std::terminate` directly.

More generally, when C++ decides that something horrifically bad has happened, such as throwing an exception during exception unwinding, its traditional response is to call `std::terminate`.

So our code to intercept all unhandled exceptions should still install a terminate handler so it can observe these “handled by terminating” exceptions.

```
#define MSVC_EXCEPTION_CODE 0xE06D7363U

std::terminate_handler previousTerminate;

LONG CALLBACK MyUnhandledExceptionFilter(
    EXCEPTION_POINTERS *ExceptionInfo)
{
    if (ExceptionInfo->ExceptionRecord.ExceptionCode ==
            MSVC_EXCEPTION_CODE) {
        CaptureDump(ExceptionInfo,
                    UnhandledException::Structured);
    } else {
        CaptureDump(ExceptionInfo,
                    UnhandledException::Cpp);
    }
    return EXCEPTION_EXECUTE_HANDLER;
}

void __cdecl my_terminate()
{
    CaptureDump(nullptr,
                UnhandledException::Cpp);
    if (previousTerminate) {
        previousTerminate();
    } else {
        std::abort();
    }
}

void InstallCustomUnhandledExceptionFilters()
{
    SetUnhandledExceptionFilter(MyUnhandledExceptionFilter);

    previousTerminate = std::set_terminate(my_terminate);
}
```

Now, when `std::terminate` is called due to mishandling of a thrown exception, the exception is considered still active and can be retrieved by `std::current_exception`. You may want to use that information when capturing the dump to extract additional information about the exception (such as its `what()`). In that case, it seems best to let all unhandled C++ exceptions go to the custom terminate handler, so it can do that work.

```
#define MSVC_EXCEPTION_CODE 0xE06D7363U

LPTOP_LEVEL_EXCEPTION_FILTER previousFilter;
std::terminate_handler previousTerminate;

LONG CALLBACK MyUnhandledExceptionFilter(
    EXCEPTION_POINTERS *ExceptionInfo)
{
    if (ExceptionInfo->ExceptionRecord.ExceptionCode ==
            MSVC_EXCEPTION_CODE) {
        return previousFilter(ExceptionInfo);
    }
    CaptureDump(ExceptionInfo,
                UnhandledException::Structured);
    return EXCEPTION_EXECUTE_HANDLER;
}

void __cdecl my_terminate()
{
    const char* what = nullptr;
    auto ptr = std::current_exception();
    if (ptr) {
        try {
            std::rethrow_exception(ptr);
        } catch (std::exception const& ex) {
            what = ex.what();
        } catch (...) {
        }
    }
    CaptureDump(nullptr,
                UnhandledException::Cpp, what);
    if (previousTerminate) {
        previousTerminate();
    } else {
        std::abort();
    }
}

void InstallCustomUnhandledExceptionFilters()
{
    previousFilter = SetUnhandledExceptionFilter(
                        MyUnhandledExceptionFilter);

    previousTerminate = std::set_terminate(my_terminate);
}
```

But wait, we're not done yet.

Some libraries let you customize how they report critical failures rather than their default behavior of failing fast. You may want to use those customization points to funnel these reports through your infrastructure. You'll have to handle these on a case-by-case basis since there is no standard for these customization points.

[1] I say “similar to” because it’s not exactly the same. The explicitly caught version forces the stack to unwind before calling `std::terminate`, but the `noexcept` version is permitted to call `std::terminate` without unwinding the stack, which is good because that lets you inspect the function’s local variables in the crash dump. We learned about this some time ago.