Can I throw a C++ exception from a structured exception?

devblogs.microsoft.com/oldnewthing/20170728-00

July 28, 2017

Raymond Chen

A customer wanted to know if it was okay to throw a C++ exception from a structured exception.

They explained that they didn't want to compile their project with the `/EHa` switch, which instructs the compiler to use the exception-handling model that catches both asynchronous (structured) exceptions as well as synchronous (C++) exceptions. In other words, the `catch` statement will catch both explicitly thrown C++ exceptions (raised by the `throw` statement) as well as exceptions generated by the operating system, either due to notifications from the CPU (such as an access violation or divide-by-zero) or explicit calls to `RaiseException`.

The customer explained that they didn't want to use `/EHa` because doing so significantly impairs compiler optimizations and results in larger code size. But on the other hand, they do want to catch the asynchronous (structured) exceptions.

So they had a fiendish plan.

Their fiendish plan is to install an unhandled exception filter which turns around and throws the C++ exception. That way, a structured exception will result in a standard C++ exception, but without the code generation penalties of the `/EHa` compiler option.

```
// This clever function is an exception filter that converts
// asynchronous exceptions (structured exception handling)
// to synchronous exceptions (C++ exceptions).

LONG WINAPI CleverConversion(
    EXCEPTION_POINTERS* ExceptionInfo)
{

    auto record = ExceptionInfo->ExceptionRecord;

    std::string message;
    ... build a message based on the exception code and
    other parameters ...

    throw std::exception(message.c_str());
}

int sample_function(int* p)
{
    try {
        printf("About to dereference the pointer %p\n", p);
        return *p;
    } catch (std::exception& e) {
        Log(e.what());
    }
    return 0;
}

int __cdecl main(int argc, char **argv)
{
    SetUnhandledExceptionFilter(CleverConversion);

    return sample_function(nullptr);
}
```

Neat trick, huh? All the benefits of `/EHa` without the overhead!

Well, except that they found that it didn't always work.

In the example above, the `catch` did catch the C++ exception, but if they took out the `printf`, then the exception was not caught.

```
int sample_function(int* p)
{
    try {
        return *p;
    } catch (std::exception& e) {
        Log(e.what());          // exception not caught!
    }
    return 0;
}
```

The customer wanted to know why the second version didn't work.

Actually the first version isn't guaranteed to work either. It happens to work because the compiler must consider the possibility that the `printf` function might throw a C++ exception. The `printf` function is not marked as `noexcept`, so the possibility is in play. (Not that you'd expect it to be marked as such, seeing as it's a C function, and C doesn't have exceptions.) When the access violation is raised as a structured exception, the `Clever-Conversion` function turns it into a C++ exception and throws it, at which point the `try` block catches it. But the `try` block is not there for the `CleverConversion` exception. It's there to catch any exceptions coming out of `printf`, and you just happened to be lucky that it caught your exception too.

In the second example, there is no call to `printf`, so the compiler says, "Well, nothing inside this `try` block can throw a C++ exception, so I can optimize out the `try/catch`." You would also have observed this behavior if there were function calls inside the `try` block, if the function calls were all to functions that were marked `noexcept` or if the compiler could prove that they didn't throw any C++ exceptions (say, because the function is inlined).

This answers the question, but let's try to look at the whole story.

1. We want to use `/EHa`.
2. But the documentation says that `/EHa` results in less efficient code. We want more efficient code, not less.
3. Aha, we found this trick that lets us convert asynchronous exceptions to synchronous ones. Now we get all the benefits of `/EHa` without any of the costs!

It looks like you found some free money on the ground, but is it really free money?

The customer seems to think that the `/EHa` option results in less efficient code simply because the compiler team is a bunch of jerks and secretly hates you.

No, that's not why the `/EHa` option results in less efficient code. The possibility that any memory access or arithmetic operation could trigger an exception significantly impairs optimization opportunities. It means that all variables must be stable at the point memory accesses occur.

Consider the following code fragment:

```
class Reminder
{
public:
    Reminder(char* message) : m_message(message) { }
    ~Reminder() { std::cout << "don't forget to "
                            << m_message << std::endl; }

    void UpdateMessage(char* message) { m_message = message; }

private:
    char* m_message;
};

void NonThrowingFunction() noexcept;
void DoSomethingElse(); // might throw

void sample_function()
{
    try {
        Reminder reminder("turn off the lights");
        if (NonThrowingFunction()) {
            reminder.UpdateMessage("feed the cat");
        }
        DoSomethingElse();
    } catch (std::exception& e) {
        Log(e.what());
    }
}
```

If compiling without `/EHa`, the compiler knows that the `NonThrowingFunction` function cannot throw a C++ exception, so it can delay the store of `reminder.m_message` to just before the call to `DoSomethingElse`. In fact, it is like to do so because it avoids a redundant store.

The pseudo-code for this function might look like this:

```
    allocate 4 bytes in local frame for reminder

l1:
    call NonThrowingFunction
    if result is zero
        load r1 = "turn off the lights"
    else
        load r1 = "feed the cat"
    endif
    store r1 to reminder.m_message
    call DoSomethingElse
l2:
    std::cout << "don't forget to "
              << r1 << std::endl;
l3:

    clean up local frame
    return

if exception occurs between l1 and l2
    std::cout << "don't forget to "
              << reminder.m_message << std::endl;
    fall through

if exception occurs between l2 and l3
    if exception is std::exception
        Log(e.what())
        goto l3
    else
        continue exception search
    endif
```

Notice that we optimized out a redundant store by delaying the initialization of `reminder`, and we enregistered `reminder. m_message` in the common code path. Delaying the initialization of `reminder` is not an optimization available to `/EHa` because of the possibility that `NonThrowingFunction` might raise an asynchronous exception that gets converted to a synchronous one:

```
    allocate 4 bytes in local frame for reminder

l0:
    // cannot delay initialization of reminder
    load r1 = "turn off the lights"
    store r1 to reminder.m_message

l1:
    call NonThrowingFunction
    if result is nonzero
        load r1 = "feed the cat"
        store r1 to reminder.m_message
    endif
    call DoSomethingElse
l2:
    std::cout << "don't forget to "
              << r1 << std::endl;
l3:

    clean up local frame
    return

if exception occurs between l1 and l2
    std::cout << "don't forget to "
              << reminder.m_message << std::endl;
    fall through

// and there is a new exception region
if exception occurs between l0 and l1, or between l2 and l3
    if exception is std::exception
        Log(e.what())
        goto l3
    else
        continue exception search
    endif
```

The extra code is necessary in order to ensure that the `reminder` variable is in a stable state before calling `NonThrowingFunction`. In general, if you turn on `/EHa`, the compiler must ensure that every object which is accessed outside the `try` block (either explicitly in code or implicitly via an unwind destructor) is stable in memory before performing any operation that could result in an asynchronous exception, such as accessing memory.

This requirement that variables be stable in memory comes at a high cost, because it not only forces redundant stores to memory, but it also prohibits various types of optimizations based on out-of-order operations.

The `CleverConversion` is basically a manual replication of what `/EHa` does, but lying to the compiler and saying, “Um, yeah, don’t worry about asynchronous exceptions.”

Observe what happens if an asynchronous exception occurs inside `NonThrowingFunction` even though you compiled without the `/EHa` flag:

We destruct the `reminder` object, which means printing the `m_message` to `std::cout`. But the non-`/EHa` version did not ensure that `reminder.m_message` was stable. Indeed, if an exception occurs inside `NonThrowingFunction`, we will try to print `reminder.m_message` anyway, even though it is an uninitialized variable.

Printing an uninitialized variable is probably not what the program intended.

So a more complete answer to the scenario is "Yes, it is technically possible to throw a C++ exception from a structured exception handler, but doing so requires that the program be compiled with `/EHa` in order to avoid undefined behavior."

And given that avoiding the `/EHa` flag was the whole purpose of the exercise, the answer to the specific scenario is, "No, this doesn't work. Your program will behave in undefined ways."

Raymond Chen

Follow