How can I handle both structured exceptions and C++ exceptions potentially coming from the same source?

devblogs.microsoft.com/oldnewthing/20200116-00

January 16, 2020

Raymond Chen

A customer had a plug-in model where the plug-in could respond to a request in three ways.

1. Return an `HRESULT`.
2. Throw a C++ exception.
3. Raise a Win32 structured exception.

The customer acknowledges that this design has made a lot of people very angry and been widely regarded as a bad move.[1] But they don't have a time machine, so they have to live with their mistakes.

The customer wanted to know whether it was appropriate to use `_set_se_translator` to convert the Win32 structured exception to a C++ exception:

```
class win32_exception { /* ... */ };

void translate_to_custom_exception(unsigned int code,
                                   PEXCEPTION_POINTERS pointers)
{
   throw win32_exception(code, get_stack_trace(pointers));
}

void InvokeCallback(CALLBACK_FUNCTION fn, /* other arguments */)
{
  auto previousTranslator =
    _set_se_translator(translate_to_custom_exception);

  try
  {
    HRESULT hr = fn(/* other arguments */);
    /* ... handle various return codes ... */
  }
  catch (const win32_exception& ewin32)
  {
    /* ... handle structured exception ... */
  }
  catch (const std::bad_alloc&)
  {
    /* ... handle low memory ... */
  }
  catch (const foo_exception& efoo)
  {
    /* ... handle foo exception ... */
  }
  /* and so on */

  _set_se_translator(previousTranslator);
}
```

The customer wanted to know whether it was okay to mix C++ and structured exceptions in this way.

Yes, it's legal. to mix C++ exceptions and structured exceptions this way. In fact, it's the stated purpose of the `_set_se_translator` function. It lets you replace the default "convert a structured exception into a C++ exception" translator with a custom one.

It may appear at first glance that you are applying a global solution to a local problem, because the `_set_se_translator` looks like it is going to change the translator for the entire process, which can result in confusing multithreading problems. But no fear, the structured exception translator is maintained on a per-thread basis. (Assuming you're using the multithreading version of the runtime library, which you'd better be using if you're multithreaded.)

However, there is still a little bit of the global/local problem, because the custom translator is in effect during the error handling phase of the function, after the plug-in has returned. The custom translator should be scoped tightly around the call to the plug-in. That way, you don't accidentally activate it if a structured exception is raised in one of the `catch` clauses, or any of the other code that isn't the call to the plug-in.

Furthermore, there's a problem if the plug-in throws an exception that wasn't explicitly caught. In that case, this code never restores the original translator. We can solve this problem by using an RAII type, so that the exception unwinder will restore the translator.

```
struct translator_guard
{
 translator_guard(_se_translator_function translator) :
    previous_translator(_set_se_translator(translator)) { }
 ~translator_guard() { _set_se_translator(previous_translator); }

 // Rule of three
 translator_guard(const translator_guard&) = delete;
 translator_guard& operator=(const translator_guard&) = delete;
private:
 _se_translator_function previous_translator;
};

void InvokeCallback(CALLBACK_FUNCTION fn, /* other arguments */)
{
  try
  {
    HRESULT hr;
    {
      translator_guard guard(translate_to_custom_exception);
      hr = fn(/* other arguments */);
    }
    /* ... handle various return codes ... */
  }
  catch (const win32_exception& ewin32)
  {
    /* ... handle structured exception ... */
  }
  catch (const std::bad_alloc&)
  {
    /* ... handle low memory ... */
  }
  catch (const foo_exception& efoo)
  {
    /* ... handle foo exception ... */
  }
  /* and so on */
}
```

Using an RAII type ensures that the translator is reset even if an exception is raised.

To avoid the bare nested scope, we could factor the the scary bits into a separate function.

```
HRESULT InvokeWithCustomExceptionTranslation(
    CALLBACK_FUNCTION fn, /* other arguments */)
{
  translator_guard guard(translate_to_custom_exception);
  return fn(/* other arguments */);
}

void InvokeCallback(CALLBACK_FUNCTION fn, /* other arguments */)
{
  try
  {
    HRESULT hr = InvokeWithCustomExceptionTranslation(
                   fn, /* other arguments */);
    /* ... handle various return codes ... */
  }
  catch (const win32_exception& ewin32)
  {
    /* ... handle structured exception ... */
  }
  catch (const std::bad_alloc&)
  {
    /* ... handle low memory ... */
  }
  catch (const foo_exception& efoo)
  {
    /* ... handle foo exception ... */
  }
  /* and so on */
}
```

The customer further noted that `_set_se_translator` requires `/EHa`, and they dutifully compiled their entire program with `/EHa`, but they were wondering if that was actually necessary. Is there an alternate design that avoids the need for `/EHa`? (Related.)

We'll take up the follow-up question next time.

Bonus chatter: I could have written

```
template<typename TLambda>
auto WithCustomExceptionTranslation(
    TLambda&& lambda)
{
  translator_guard guard(translate_to_custom_exception);
  return lambda();
}
```

and call it from `InvokeCallback` like this:

```
HRESULT hr = WithCustomExceptionTranslation(
             [&] { return fn(/* other arguments */); });
```

Is this a good thing or a bad thing?

[1] I suspect that this was not the original design for the plug-in model, but people abused the plug-in model so much that they ended up forced to support it, for compatibility reasons.

Raymond Chen

Follow