What can I do if I want to throw a C++ exception from my InitOnce callback?

devblogs.microsoft.com/oldnewthing/20170406-00

April 6, 2017

Raymond Chen

Suppose you want to use the `InitOnceExecuteOnce` function to perform one-time initialization, but your initialization function might throw a C++ exception. We know that this is not allowed because you don't control all the frames the exception is being thrown across, so what are your options?

The naïve solution is to catch the exception before it escapes your callback, and then rethrow it on the other side.

```
void Sample()
{
    struct State {
        std::exception_ptr p;
        // other members you want to access from the lambda
    } state;
    if (!InitOnceExecuteOnce(&g_InitOnce,
          [](PINIT_ONCE InitOnce, void* Parameter, void** Context)
          -> BOOL
          {
              auto s = reinterpret_cast<State*>(Parameter);
              try {
                  init_stuff();
                  return TRUE;
              } catch (std::exception& e) {
                  s->p = std::current_exception();
                  return FALSE;
              }
          }, &state, nullptr)) {
        // Failed due to exception.  Rethrow now that we are
        // safely outside the InitOnceExecuteOnce function.
        std::rethrow_exception(state.p);
    }
}
```

A less cumbersome solution is to use synchronous two-phase initialization:

```
void Sample()
{
    void* result;
    BOOL pending;
    if (!InitOnceBeginInitialize(&g_InitOnce, 0,
                                 &pending, &result)) {
        if (pending) {
            try {
                init_stuff();
            } catch (...) {
                InitOnceComplete(&g_InitOnce,
                                 INIT_ONCE_INIT_FAILED, result);
                throw;
            }
            InitOnceComplete(&g_InitOnce, 0, result);
        }
    }
}
```

Synchronous two-phase initialization performs the initialization inline rather than in a callback, which saves you the trouble of having to save the exception in one place and rethrow it in another place. You can just tell the InitOnce engine that initialization failed and then allow the exception to propagate.

You might decide to wrap this pattern inside an RAII type.

```
class InitOnceGuard
{
public:
    InitOnceGuard(PINIT_ONCE initOnce) :
        m_initOnce(initOnce),
        m_success(InitOnceBeginInitialize(initOnce, 0, &m_pending, nullptr)),
        m_status(INIT_ONCE_INIT_FAILED)
    {
    }

    ~InitOnceGuard()
    {
        if (NeedInitialization()) InitOnceComplete(m_initOnce, m_status, nullptr);
    }

    InitOnceGuard(const InitOnceGuard&) = delete;
    InitOnceGuard(InitOnceGuard&&) = delete;
    InitOnceGuard& operator=(const InitOnceGuard&) = delete;
    InitOnceGuard& operator=(const InitOnceGuard&&) = delete;

    bool NeedInitialization() { return m_success && m_pending; }

    // If you don't Complete, then the guard assumes that initialization failed.
    void Complete() { m_status = 0; }

private:
    PINIT_ONCE m_initOnce;
    bool m_success;
    bool m_pending;
    DWORD m_status;
}

void Sample()
{
    InitOnceGuard guard(&g_InitOnce);
    if (guard.NeedInitialization()) {
        init_stuff();
        guard.Complete();
    }
}
```

If the `guard` destructs without ever being `Complete`d, either because of an exception, or because the caller decided that initialization failed in an unexceptional way, then the destructor will tell the InitOnce engine that initialization failed. This will unblock any other threads that are waiting for initialization to complete and allow them to give it a try.

If the `guard` is `Complete`d, then its destructor tells the InitOnce engine that initialization was successful.

After thinking about all that, you might realize that the fact that you're throwing C++ exceptions means that you're already committed to C++, so you may as well go all in: Use `std::call_once` or C++ static locals. These are part of the C++ standard and are fully exception-aware. Of course, it assumes that all the frames you are throwing across came from the same C++ compiler.

Raymond Chen

Follow