The case of the invalid handle despite being managed by an RAII type, part 2

devblogs.microsoft.com/oldnewthing/20230616-00

June 16, 2023

Raymond Chen

Last time, we investigated the case of the invalid handle despite being managed by an RAII type, and we identified that the problem was a use-after-free. So how do we fix the problem?

The minimum fix is to stop the tracker and wait for it to be destroyed before assessing the results.

```
// Stop the tracker and wait for it to be destroyed.
tracker.reset();
results.WaitForDestroyed();
⇅
// Fail if no qualifying widget was found
THROW_HR_IF(HRESULT_FROM_WIN32(ERROR_NOT_FOUND),
            results.GetWidgetId() == INVALID_WIDGET_ID);
```

This fixes the problem, but it's a band-aid. Really, we should make this type of mistake harder to make in the first place.

One idea is to put the `destroyedEvent.wait()` in the `FirstWidgetResults` destructor, so that no matter what happens, we always wait for the tracker to finish its callbacks. But we have to be careful to do this only if the tracker was *started*. So maybe we can do this:

```
struct FirstWidgetResults
{
  ~FirstWidgetResults()
  {
    if (started) destroyedEvent.wait();
  }

  int GetWidgetId() { return widgetId; }
  void WaitForResults()
  {
    started = true;
    readyEvent.wait();
  }
  // void WaitForDestroyed() { destroyedEvent.wait(); }

  static void CALLBACK OnCallback(
      int reason, int id, void* context)
  {
    auto self = (MyTrackerResults*)context;

    switch (reason) {
    case WidgetFound:
      if (self->widgetId == INVALID_WIDGET_ID) {
        self->widgetId = id;
        self->readyEvent.Set();
      }
      break;

    case InitialScanComplete:
      self->readyEvent.Set();
      break;

    case TrackerDestroyed:
      self->destroyedEvent.Set();
      break;
    }
  }

private:
  int widgetId = INVALID_WIDGET_ID;
  bool started = false;

  wil::unique_event readyEvent{ wil::EventOptions::None };
  wil::unique_event destroyedEvent{ wil::EventOptions::None };
};

Widget FindWidgetWithEnabledAdapter()
{
  TrackerOptions options;
  /* ... code to set the options ... */

  FirstWidgetResults results;
```

```
  // Start the tracker (or die trying)
  unique_tracker tracker;
  THROW_IF_FAILED(CreateTracker(
        &options,
        FirstWidgetResults::OnCallback,
        &results,
        &tracker));

  // Wait for results to arrive
  results.WaitForResults();

  // Stop the tracker // and wait for it to be destroyed.
  tracker.reset();
  // results.WaitForDestroyed();

  // Fail if no qualifying widget was found
  THROW_HR_IF(HRESULT_FROM_WIN32(ERROR_NOT_FOUND),
              results.widgetId == INVALID_WIDGET_ID);

  // Return the widget that we found.
  return WidgetFromId(results.GetWidgetId());
}
```

Once nice thing about this new version that you also don't have to worry about making sure to stop the tracker before calling `WaitForDestroyed()`. The `WaitForDestroyed()` always happens last because the `FirstWidgetResults` was constructed first, and local objects destruct in reverse order of construction. Even if you forget to stop the tracker, it will stop on its own at destruction. (The only reason to stop it earlier is performance: It allows you to do other work, like converting the Widget ID to a Widget, in parallel with the teardown of the tracker.)

But really, there's no reason for the caller to have access to the tracker. They don't care about the tracker. They care only about the results. The way the code was originally written, the only reason the caller has the tracker is so that the caller can shut it down. But we can make that the responsibility of the `FirstWidgetResults`.

```
struct FirstWidgetResults
{
  ~FirstWidgetResults()
  {
    if (started) destroyedEvent.wait();
  }

  int GetWidgetId() { return widgetId; }

  void RunQuery(TrackerOptions* options)
  {
    // Start the tracker (or die trying)
    unique_tracker tracker;
    THROW_IF_FAILED(CreateTracker(
        &options,
        FirstWidgetResults::OnCallback,
        &results,
        &tracker));

    started = true;
    readyEvent.wait();
  }
private:
  static void CALLBACK OnCallback(
      int reason, int id, void* context)
  {
    auto self = (MyTrackerResults*)context;

    switch (reason) {
    case WidgetFound:
      if (self->widgetId == INVALID_WIDGET_ID) {
        self->widgetId = id;
        self->readyEvent.Set();
      }
      break;

    case InitialScanComplete:
      self->readyEvent.Set();
      break;

    case TrackerDestroyed:
      self->destroyedEvent.Set();
      break;
    }
  }

private:
  int widgetId = INVALID_WIDGET_ID;
  bool started = false;

  wil::unique_event readyEvent{ wil::EventOptions::None };
  wil::unique_event destroyedEvent{ wil::EventOptions::None };
```

```
};

Widget FindWidgetWithEnabledAdapter()
{
  TrackerOptions options;
  /* ... code to set the options ... */

  FirstWidgetResults results;

  // Start the tracker (or die trying) and get the results
  results.RunQuery(&options);

  // Wait for results to arrive
  // results.WaitForResults();

  // Stop the tracker
  // tracker.reset();

  // Fail if no qualifying widget was found
  THROW_HR_IF(HRESULT_FROM_WIN32(ERROR_NOT_FOUND),
              results.GetWidgetId() == INVALID_WIDGET_ID);

  // Return the widget that we found.
  return WidgetFromId(results.GetWidgetId());
}
```

The `FirstWidgetResults` class manages the tracker. This also implicitly does the “early tracker destruction” optimization, because the tracker destructs as part of returning from `RunQuery`, long before the `FirstWidgetResults` destructs.

Finally, we can put the `RunQuery` into the constructor. This avoids the need for a `started` state variable.

```
struct FirstWidgetResults
{
  ~FirstWidgetResults()
  {
    // if (started)
    destroyedEvent.wait();
  }

  int GetWidgetId() { return widgetId; }

  FirstWidgetResults(TrackerOptions* options)
  {
    // Start the tracker (or die trying)
    unique_tracker tracker;

    // Throwing from a constructor bypasses the destructor
    THROW_IF_FAILED(CreateTracker(
        &options,
        FirstWidgetResults::OnCallback,
        &results,
        &tracker));

    // started = true;
    readyEvent.wait();
  }

  ...
  // bool started = false;
  ...
};

Widget FindWidgetWithEnabledAdapter()
{
  TrackerOptions options;
  /* ... code to set the options ... */

  FirstWidgetResults results(&options);

  // Fail if no qualifying widget was found
  THROW_HR_IF(HRESULT_FROM_WIN32(ERROR_NOT_FOUND),
              results.GetWidgetId() == INVALID_WIDGET_ID);

  // Return the widget that we found.
  return WidgetFromId(results.GetWidgetId());
}
```

This takes advantage of the fact that throwing from a class's constructor bypasses the class's destructor, because you can't destruct something that hasn't fully constructed. This aspect of the C++ language is often a gotcha, but here we are taking advantage of it: If `CreateTracker` fails, then the tracker never started, so we don't want to wait for it, and bypassing the destructor is exactly what we want.

Now, this “bypassing the destructor” behavior is rather subtle, so I can see why you wouldn’t want to take advantage of it, especially since you can introduce bugs if you add more code to the constructor that could throw after the tracker has started, so this last transformation may not be something you’re comfortable with.

And that’s fine. I’m not sure I’m comfortable with it either.