Spotting problems with destructors for C++ temporaries

devblogs.microsoft.com/oldnewthing/20190429-00

April 29, 2019

Raymond Chen

Consider the unique_handle. It specializes `std:: unique_ ptr` to support Windows kernel handles. It lets you get all the niceties of `std:: unique_ ptr` with just a handful of lines of code.

But then you have the problem of interoperating with the rest of the system. For example, how would you use a `unique_ handle` to receive the result of a duplication?

```
unique_handle originalHandle = ...;
unique_handle duplicateHandle;

if (DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), &duplicateHandle,
      0, FALSE, DUPLICATE_SAME_ACCESS)) { ... }
```

This doesn't compile because the `operator&` for a `unique_ ptr` doesn't give you a pointer to the inner raw pointer. You'll have to perform the operation in two steps.

```
HANDLE rawDuplicateHandle = nullptr;

// First, get a raw handle as the duplicate.
auto result = DuplicateHandle(
    GetCurrentProcess(), originalHandle.get(),
    GetCurrentProcess(), &rawDuplicateHandle,
    0, FALSE, DUPLICATE_SAME_ACCESS);

// Then set it into the smart pointer.
duplicateHandle.reset(rawDuplicateHandle);

// Then see if it worked.
if (result) { ... }
```

We could tune this a little:

```
HANDLE rawDuplicateHandle;

// Out with the old.
duplicateHandle.reset();

// Try to get a new handle.
if (DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), &rawDuplicateHandle,
      0, FALSE, DUPLICATE_SAME_ACCESS)) {

  // Save the new handle back into the smart pointer.
  duplicateHandle.reset(rawDuplicateHandle);

  ...
}
```

But the underlying issue remains: Bridging the gap between the C++ `unique_ptr` and the system function that wants you to pass the address of a `HANDLE`.

You might decide to create a helper class whose job is to encapsulate this two-step dance, acting as a proxy between the raw handle and the smart pointer.

```
struct handle_proxy
{
  handle_proxy(unique_handle& output)
  : m_output(output) { }

  ~handle_proxy() { m_output.reset(m_rawHandle); }

  HANDLE* addressof() { return &m_rawHandle; }

  // Not copyable, not movable.
  handle_proxy(const handle_proxy&) = delete;
  handle_proxy& operator=(const handle_proxy&) = delete;

  unique_handle& m_output;
  HANDLE m_rawHandle = nullptr;
};
```

This proxy object lets you pass a `HANDLE*` to functions that return a handle through a pointer, and when the proxy is destructed, it transfers the raw handle into the smart pointer.

```
DuplicateHandle(
    GetCurrentProcess(), originalHandle.get(),
    GetCurrentProcess(), handle_proxy(duplicateHandle).addressof(),
    0, FALSE, DUPLICATE_SAME_ACCESS);
```

What's happening here is that we create a temporary `handle_proxy` object to pass to the `DuplicateHandle` function. This temporary object remembers the `unique_handle` that it is proxying for, and produces the address of a plain old `HANDLE` which is what gets passed to the to the `DuplicateHandle` function. After the `DuplicateHandle` function returns, the temporary is destructed, and the destructor takes the raw handle in `m_rawHandle` and puts it into the smart pointer.

As a convenience, you could add a conversion operator to save people the hassle of having to say `addressof` all over the place.

```
struct handle_proxy
{
  handle_proxy(unique_handle& output)
  : m_output(output) { }

  ~handle_proxy() { m_output.reset(m_rawHandle); }

  HANDLE* addressof() { return &m_rawHandle; }
  operator HANDLE*() { return addressof(); }

  // Not copyable, not movable.
  handle_proxy(const handle_proxy&) = delete;
  handle_proxy& operator=(const handle_proxy&) = delete;

  unique_handle& m_output;
  HANDLE m_rawHandle = nullptr;
};

DuplicateHandle(
    GetCurrentProcess(), originalHandle.get(),
    GetCurrentProcess(), handle_proxy(duplicateHandle),
    0, FALSE, DUPLICATE_SAME_ACCESS);
```

Everything's looking great, until somebody does this:

```
// Try to duplicate the handle for EVENT_MODIFY_STATE access
// and reset it.
if (DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), handle_proxy(duplicateHandle),
      EVENT_MODIFY_STATE, FALSE, 0) &&
    ResetEvent(duplicateHandle.get()) { ... }
```

Do you see the problem?

The C++ rules for temporary objects is that they are destructed at the end of the "full expression" that contains them. This means that the temporary `handle_ proxy` object doesn't get destructed until the entire expression inside the `if` statement has been evaluated, which means that the `ResetEvent` happens before the proxy's destructor can transfer the raw pointer into the smart pointer.

1. Create temporary `handle_ proxy`.

2. Convert it to a `HANDLE*`.
3. Call `DuplicateHandle`.
4. Assuming `DuplicateHandle` succeeds, call `ResetEvent` with the `duplicateHandle`.
5. Destruct the `handle_ proxy`, which copies the raw handle into `duplicateHandle`.

We want step 5 to happen before step 4, but the C++ rules for destruction of temporary objects forces the proxy to linger until after the expression has been evaluated.

What can we do?

One option is to introduce a dreaded macro which forces temporaries to be destructed prematurely.

```
#define DESTRUCT_TEMPORARIES(v) [&]() { return (v); }()
```

This macro wraps the argument inside an immediately-evaluated lambda and propagates the value of the expression. This doesn't seem to accomplish anything, but what it does is pull `v` into its own full expression, thereby forcing its temporaries to be destructed immediately after its evaluation.

```
if (DESTRUCT_TEMPORARIES(DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), handle_proxy(duplicateHandle),
      EVENT_MODIFY_STATE, FALSE, 0)) &&
    ResetEvent(duplicateHandle.get()) { ... }
```

This is ugly for multiple reasons. One is that it encourages passing multi-line entities to macros. Second, it's a macro used for something other than `#ifdef`.

Another option is to manage the transfer explicitly.

```
struct handle_proxy
{
  handle_proxy(unique_handle& output)
  : m_output(std::addressof(output)) { }

  ~handle_proxy() { transfer(); }

  void transfer()
  {
    if (m_output) {
      std::exchange(m_output, {})->reset(m_rawHandle);
    }
  }

  HANDLE* addressof() { return &m_rawHandle; }
  operator HANDLE*() { return addressof(); }

  // Not copyable.
  handle_proxy(const handle_proxy&) = delete;
  handle_proxy& operator=(const handle_proxy&) = delete;

  // Movable.
  handle_proxy(handle_proxy&& other)
  : m_output(std::exchange(other.m_output, {})),
    m_rawHandle(std::exchange(other.m_rawHandle, {})) { }

  handle_proxy& operator=(handle_proxy&& other)
  {
    transfer();
    m_output = std::exchange(other.m_output, {});
    m_rawHandle = std::exchange(other.m_rawHandle, {});
  }

  unique_handle* m_output;
  HANDLE m_rawHandle = nullptr;
};
```

In this version, we capture a raw pointer to the `unique_ handle` so that we have a way to keep track of whether the result has been transferred or not: If the pointer is `nullptr`, then the transfer has already taken place.

Now that we have a way to say "Nothing to transfer," we can add a move constructor and move assignment operator, both of which leave the source in the "Nothing to do" state.

You can continue to use this version of the `handle_proxy` the same as the previous version, or you can opt to trigger the transfer early:

```
if (auto proxy = handle_proxy(duplicateHandle);
    DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), proxy,
      EVENT_MODIFY_STATE, FALSE, 0)) &&
    (proxy.transfer(), ResetEvent(duplicateHandle.get())) { ... }
```

This is still pretty ugly. You can clean it up a little by having the transfer function also return the handle that it transferred.

```
struct handle_proxy
{
  ...
  HANDLE transfer()
  {
    if (m_output) {
      std::exchange(m_output, {})->reset(m_rawHandle);
    }
    return m_rawHandle;
  }
  ...
};

if (auto proxy = handle_proxy(duplicateHandle);
    DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), proxy,
      EVENT_MODIFY_STATE, FALSE, 0)) &&
    ResetEvent(proxy.transfer())) { ... }
```

It’s a little less ugly, but also more puzzling.

Perhaps the way out is simply to split it into two expressions.

```
if (DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), handle_proxy(duplicateHandle),
      EVENT_MODIFY_STATE, FALSE, 0)) {

  // Test separately to give handly_proxy's destructor a chance to
  // put the result into duplicateHandle before we read it out.
  if (ResetEvent(duplicateHandle.get())) { ... }
}
```

If you are willing to wrap every system function you need to interact with, you could add a wrapper that returns a `unique_ handle` directly.

```
unique_handle DuplicateAndReturnHandle(
    HANDLE sourceProcess,
    HANDLE sourceHandle,
    HANDLE targetProcess,
    DWORD desiredAccess,
    BOOL  inheritHandle,
    DWORD options)
{
  HANDLE targetHandle;
  if (DuplicateHandle(sourceProcess, sourceHandle,
                      targetProcess, &targetHandle,
                      desiredAccess, inheritHandle, options)) {
    return unique_handle{ targetHandle };
  }
  return {};
}

if ((duplicateHandle = DuplicateAndReturnHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), EVENT_MODIFY_STATE, FALSE, 0)) &&
  ResetEvent(duplicateHandle.get())) { ... }
}
```

The downside of this is that you have an entire library of wrapper functions you'll have to maintain. And if the function is like `RegOpenKeyEx` and returns information in addition to the handle, you'll have to put it into a `std:: tuple` or a `std:: variant`, which is another level of bother.

Consider the unique_handle. It specializes `std:: unique_ ptr` to support Windows kernel handles. It lets you get all the niceties of `std:: unique_ ptr` with just a handful of lines of code.

But then you have the problem of interoperating with the rest of the system. For example, how would you use a `unique_ handle` to receive the result of a duplication?

```
unique_handle originalHandle = ...;
unique_handle duplicateHandle;

if (DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), &duplicateHandle,
      0, FALSE, DUPLICATE_SAME_ACCESS)) { ... }
```

This doesn't compile because the `operator&` for a `unique_ ptr` doesn't give you a pointer to the inner raw pointer. You'll have to perform the operation in two steps.

```
HANDLE rawDuplicateHandle = nullptr;

// First, get a raw handle as the duplicate.
auto result = DuplicateHandle(
    GetCurrentProcess(), originalHandle.get(),
    GetCurrentProcess(), &rawDuplicateHandle,
    0, FALSE, DUPLICATE_SAME_ACCESS);

// Then set it into the smart pointer.
duplicateHandle.reset(rawDuplicateHandle);

// Then see if it worked.
if (result) { ... }
```

We could tune this a little:

```
HANDLE rawDuplicateHandle;

// Out with the old.
duplicateHandle.reset();

// Try to get a new handle.
if (DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), &rawDuplicateHandle,
      0, FALSE, DUPLICATE_SAME_ACCESS)) {

  // Save the new handle back into the smart pointer.
  duplicateHandle.reset(rawDuplicateHandle);

  ...
}
```

But the underlying issue remains: Bridging the gap between the C++ `unique_ptr` and the system function that wants you to pass the address of a `HANDLE`.

You might decide to create a helper class whose job is to encapsulate this two-step dance, acting as a proxy between the raw handle and the smart pointer.

```
struct handle_proxy
{
  handle_proxy(unique_handle& output)
  : m_output(output) { }

  ~handle_proxy() { m_output.reset(m_rawHandle); }

  HANDLE* addressof() { return &m_rawHandle; }

  // Not copyable, not movable.
  handle_proxy(const handle_proxy&) = delete;
  handle_proxy& operator=(const handle_proxy&) = delete;

  unique_handle& m_output;
  HANDLE m_rawHandle = nullptr;
};
```

This proxy object lets you pass a `HANDLE*` to functions that return a handle through a pointer, and when the proxy is destructed, it transfers the raw handle into the smart pointer.

```
DuplicateHandle(
    GetCurrentProcess(), originalHandle.get(),
    GetCurrentProcess(), handle_proxy(duplicateHandle).addressof(),
    0, FALSE, DUPLICATE_SAME_ACCESS);
```

What's happening here is that we create a temporary `handle_proxy` object to pass to the `DuplicateHandle` function. This temporary object remembers the `unique_handle` that it is proxying for, and produces the address of a plain old `HANDLE` which is what gets passed to the to the `DuplicateHandle` function. After the `DuplicateHandle` function returns, the temporary is destructed, and the destructor takes the raw handle in `m_rawHandle` and puts it into the smart pointer.

As a convenience, you could add a conversion operator to save people the hassle of having to say `addressof` all over the place.

```
struct handle_proxy
{
  handle_proxy(unique_handle& output)
  : m_output(output) { }

  ~handle_proxy() { m_output.reset(m_rawHandle); }

  HANDLE* addressof() { return &m_rawHandle; }
  operator HANDLE*() { return addressof(); }

  // Not copyable, not movable.
  handle_proxy(const handle_proxy&) = delete;
  handle_proxy& operator=(const handle_proxy&) = delete;

  unique_handle& m_output;
  HANDLE m_rawHandle = nullptr;
};

DuplicateHandle(
    GetCurrentProcess(), originalHandle.get(),
    GetCurrentProcess(), handle_proxy(duplicateHandle),
    0, FALSE, DUPLICATE_SAME_ACCESS);
```

Everything's looking great, until somebody does this:

```
// Try to duplicate the handle for EVENT_MODIFY_STATE access
// and reset it.
if (DuplicateHandle(
        GetCurrentProcess(), originalHandle.get(),
        GetCurrentProcess(), handle_proxy(duplicateHandle),
        EVENT_MODIFY_STATE, FALSE, 0) &&
    ResetEvent(duplicateHandle.get()) { ... }
```

Do you see the problem?

The C++ rules for temporary objects is that they are destructed at the end of the "full expression" that contains them. This means that the temporary `handle_ proxy` object doesn't get destructed until the entire expression inside the `if` statement has been evaluated, which means that the `ResetEvent` happens before the proxy's destructor can transfer the raw pointer into the smart pointer.

1. Create temporary `handle_ proxy` .
2. Convert it to a `HANDLE*` .
3. Call `DuplicateHandle` .
4. Assuming `DuplicateHandle` succeeds, call `ResetEvent` with the `duplicate-Handle` .
5. Destruct the `handle_ proxy` , which copies the raw handle into `duplicateHandle` .

We want step 5 to happen before step 4, but the C++ rules for destruction of temporary objects forces the proxy to linger until after the expression has been evaluated.

What can we do?

One option is to introduce a dreaded macro which forces temporaries to be destructed prematurely.

```
#define DESTRUCT_TEMPORARIES(v) [&]() { return (v); }()
```

This macro wraps the argument inside an immediately-evaluated lambda and propagates the value of the expression. This doesn't seem to accomplish anything, but what it does is pull `v` into its own full expression, thereby forcing its temporaries to be destructed immediately after its evaluation.

```
if (DESTRUCT_TEMPORARIES(DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), handle_proxy(duplicateHandle),
      EVENT_MODIFY_STATE, FALSE, 0)) &&
    ResetEvent(duplicateHandle.get()) { ... }
```

This is ugly for multiple reasons. One is that it encourages passing multi-line entities to macros. Second, it's a macro used for something other than `#ifdef`.

Another option is to manage the transfer explicitly.

```
struct handle_proxy
{
  handle_proxy(unique_handle& output)
  : m_output(std::addressof(output)) { }

  ~handle_proxy() { transfer(); }

  void transfer()
  {
    if (m_output) {
      std::exchange(m_output, {})->reset(m_rawHandle);
    }
  }

  HANDLE* addressof() { return &m_rawHandle; }
  operator HANDLE*() { return addressof(); }

  // Not copyable.
  handle_proxy(const handle_proxy&) = delete;
  handle_proxy& operator=(const handle_proxy&) = delete;

  // Movable.
  handle_proxy(handle_proxy&& other)
  : m_output(std::exchange(other.m_output, {})),
    m_rawHandle(std::exchange(other.m_rawHandle, {})) { }

  handle_proxy& operator=(handle_proxy&& other)
  {
    transfer();
    m_output = std::exchange(other.m_output, {});
    m_rawHandle = std::exchange(other.m_rawHandle, {});
  }

  unique_handle* m_output;
  HANDLE m_rawHandle = nullptr;
};
```

In this version, we capture a raw pointer to the `unique_ handle` so that we have a way to keep track of whether the result has been transferred or not: If the pointer is `nullptr`, then the transfer has already taken place.

Now that we have a way to say "Nothing to transfer," we can add a move constructor and move assignment operator, both of which leave the source in the "Nothing to do" state.

You can continue to use this version of the `handle_proxy` the same as the previous version, or you can opt to trigger the transfer early:

```
if (auto proxy = handle_proxy(duplicateHandle);
    DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), proxy,
      EVENT_MODIFY_STATE, FALSE, 0)) &&
    (proxy.transfer(), ResetEvent(duplicateHandle.get())) { ... }
```

This is still pretty ugly. You can clean it up a little by having the transfer function also return the handle that it transferred.

```
struct handle_proxy
{
  ...
  HANDLE transfer()
  {
    if (m_output) {
      std::exchange(m_output, {})->reset(m_rawHandle);
    }
    return m_rawHandle;
  }
  ...
};

if (auto proxy = handle_proxy(duplicateHandle);
    DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), proxy,
      EVENT_MODIFY_STATE, FALSE, 0)) &&
    ResetEvent(proxy.transfer())) { ... }
```

It’s a little less ugly, but also more puzzling.

Perhaps the way out is simply to split it into two expressions.

```
if (DuplicateHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), handle_proxy(duplicateHandle),
      EVENT_MODIFY_STATE, FALSE, 0)) {

  // Test separately to give handly_proxy's destructor a chance to
  // put the result into duplicateHandle before we read it out.
  if (ResetEvent(duplicateHandle.get())) { ... }
}
```

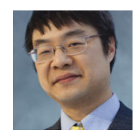
If you are willing to wrap every system function you need to interact with, you could add a wrapper that returns a `unique_ handle` directly.

```
unique_handle DuplicateAndReturnHandle(
    HANDLE sourceProcess,
    HANDLE sourceHandle,
    HANDLE targetProcess,
    DWORD desiredAccess,
    BOOL  inheritHandle,
    DWORD options)
{
  HANDLE targetHandle;
  if (DuplicateHandle(sourceProcess, sourceHandle,
                      targetProcess, &targetHandle,
                      desiredAccess, inheritHandle, options)) {
    return unique_handle{ targetHandle };
  }
  return {};
}

if ((duplicateHandle = DuplicateAndReturnHandle(
      GetCurrentProcess(), originalHandle.get(),
      GetCurrentProcess(), EVENT_MODIFY_STATE, FALSE, 0)) &&
  ResetEvent(duplicateHandle.get())) { ... }
}
```

The downside of this is that you have an entire library of wrapper functions you'll have to maintain. And if the function is like `RegOpenKeyEx` and returns information in addition to the handle, you'll have to put it into a `std:: tuple` or a `std:: variant`, which is another level of bother.

Raymond Chen



Follow