Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Prefer Using Futures or Callbacks to Communicate Asynchronous Results

Handling asynchronous communication back to the caller

August 12, 2010
URL:http://www.drdobbs.com/cpp/prefer-using-futures-or-callbacks-to-com/226700179

Active objects offer an important abstraction above raw threads. In a previous article, we saw how active objects let us hide the private thread, deterministically organize the thread's work, isolate its private data, express asynchronous messages as ordinary method calls, and express thread/task lifetimes as object lifetimes so that we can manage both using the same normal language features. [1]

What we didn't cover, however, was how to handle methods' return values and/or "out" parameters, and other kinds of communication back to the caller. This time, we'll answer the following questions:

- How should we express return values and out parameters from an asynchronous function, including an active object method?
- How should we give back multiple partial results, such as partial computations or even just "percent done" progress information?
- Which mechanisms are suited to callers that want to "pull" results, as opposed to having the callee "push" the results back proactively? And how can "pull" be converted to "push" when we need it?

Let's dig in.

Getting Results: Return Values and "Out" Parameters

First, let's recall the active object example we introduced last time.

We have a GUI thread that must stay responsive, and to keep it ready to handle new messages we have to move "save this document," "print this document," and any other significant work off the responsive thread to run asynchronously somewhere else. One way to do that is to have a background worker thread that handles the saving and print rendering work. We feed the work to the background thread by sending it asynchronous messages that contain the work to be performed; the messages are queued up if the worker thread is already busy, and then executed sequentially on the background worker thread.

The following code expresses the background worker using a `Backgrounder` class that follows the active object pattern. The code we'll show uses C++0x syntax, but can be translated directly into other popular threading environments such as those provided by Java, .NET, and Pthreads (see [1] for a discussion of the pattern and translating the code to other environments).

The `Active` helper member encapsulates a private thread and message queue, and each `Backgrounder` method call simply captures its parameters and its body (both conveniently automated by using lambda function syntax) and Send that as an asynchronous message that's fired off to be enqueued and later executed on the private thread:

```
// Baseline example
class Backgrounder {
public:
  void Save( string filename ) { a.Send( [=] {
    // … do the saving work …
  } ); }

  void Print( Data& data ) { a.Send( [=, &data] {
    do {
      // … do the printing work for another piece of the data …
    } while( /* not done formatting the data */ );
  } ); }

private:
  PrivateData somePrivateStateAcrossCalls;
  Active a;    // encapsulates a private thread, and
};    // pumps method calls as messages
```

The GUI thread instantiates a single `Backgrounder` object (and therefore a single background worker thread), which might be used from the GUI thread as follows:

```
class MyGUI {
public:
  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    backgrounder.Save( filename );
      // this fires off an asynchronous message
  } // and then we return immediately to the caller

  // …

private:
  Backgrounder backgrounder;
```

```
};
```

This illustrates the ability to treat asynchronous messages like normal method calls and everything is all very nice, but you might have noticed that the `Save` and `Print` methods (in)conveniently don't have any return value or "out" parameters, don't report their progress or intermediate results, nor communicate back to the caller at all. What should we do if we actually want to see full or partial results?

Option 1: Return a Future (Caller Naturally "Pulls")

First, let's deal with just the return value and output parameters, which should be somehow communicated back to the caller at the end of the asynchronous method. To keep the code simple, we'll focus on the primary return value; output parameters are conceptually just additional return values and can be handled the same way.

For an asynchronous method call, we want to express its return value as an asynchronous result. The default tool to use for an asynchronous value is a "future" (see [2]). To keep the initial example simple, let's say that `Save` just wants to return whether it succeeded or failed, by returning a bool:

```
// Example 1: Return value, using a future
class Backgrounder {
public:
  future<bool> Save( string filename ) {
    // Make a future (to be waited for by the caller)
    // connected to a promise (to be filled in by the callee)
    auto p = make_shared<promise<bool>>();
    future<bool> ret = p->get_future();
    a.Send( [=] {
      // … do the saving work …
      p->set_value( didItSucceed() ? true : false );
    } );
    return ret;
  }
```

(C++0x-specific note: Why are we holding the promise by reference-counted smart pointer? Because promise is a move-only type and C++ lambdas do not yet support move-capture, only capture-by-value and capture-by-reference. One simple solution is to hold the promise by `shared_ptr`, and copy that.)

Now the caller can wait for the "future":

```
future<bool> result = backgrounder.Save( filename );
…
… this code can run concurrently with Save()
…
Use( result.get() );  // block if necessary until result is available
```

This works, and returning a "future" is generally a useful mechanism.

However, notice that waiting for a "future" is inherently a "pull" operation; that is, the caller has to ask for the result when it's needed. For callers who want to find out if the result is ready without blocking if it isn't, "future" types typically provide a status method like `result.is_ready()` for the caller to check without blocking, which he can do in a loop and then sleep in between calls — that's still a form of polling loop, but at least it's better than burning CPU cycles with outright busy-waiting.

So, although the caller isn't forced to busy-wait, the onus is still on him to act to "pull" the value. What can we do if instead the caller wants a "push" notification sent to him proactively when the result is available? Let's consider two ways, which we'll call Options 2 and 3.

Option 2: Return a Future (Caller Converts "Pull" Into "Push")

The "pull" model is great for many uses, but the example caller we saw above is a must-stay-responsive GUI thread. That kind of caller certainly doesn't want to wait for the "future" on any GUI thread method, because responsive threads must not block or stop processing new events and messages. There are workarounds, but they're not ideal: For example, it's possible for the GUI thread to remember there's a "future" and check it on each event that gets processed, and to make sure it sees it soon enough it can generate extra timer events to be woken up just so it can check the "future" — but that seems (and is) a lot harder than it should be.

Given that a responsive thread like a GUI thread is already event-driven, ideally we would like it to be able to receive the result as a new queued event message that it can naturally respond to just like anything else.

Option 2 is to have the caller do the work to convert "pull" to "push." In this model, the callee still returns a "future" as in Option 1, and it's up to the caller turn it into a proactive result. How can the caller do that? One way is to launch a one-off asynchronous operation that just waits for the "future" and then generates a notification from the result. Here's an example:

```
// Example 2(a): Calling code, taking result as a future
// and converting it into an event-driven notification
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    shared_future<bool> result;
    result = backgrounder.Save( filename );
    // queue up a continuation to notify ourselves of the
    // result once it's available
    async( [=] { SendSaveComplete( result.get() ); } );
  }
```

```
  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

The statement `SendSaveComplete( result->get() );` does two things: First, it executes `result->get()`, which blocks if necessary to wait for the result to be available. Then, and only then, it calls `SendSaveComplete`; in this case, an asynchronous method that when executed ends up calling `OnSaveComplete` and passes the available result. (C++0x-specific note: Like promises, ordinary "futures" are intended to be unique and therefore not copyable, but are move-only, so again we use the `shared_ptr` workaround to enable copying the result into the lambda for later use.)

Option 2 Variant: ContinueWith

As written, the Example 2(a) code has two disadvantages:

- First, it spins up (and ties up) a thread just to keep the asynchronous operation alive, which is potentially unnecessary because the first thing the asynchronous operation does is go idle until something happens.
- Second, it incurs an extra wakeup, because when the "future" result is available, we need to wake up the asynchronous helper operation's thread and continue there.

Some threading platforms offer a "future-like" type that has a `ContinueWith`-style method to avoid this overhead; for example, see `.NET's Task<T>`. [3] The idea is that `ContinueWith` takes a continuation to be executed once the thread that fills the "future" makes the "future" ready, and the continuation can be executed on that same target thread.

Tacking the continuation onto the "future-generating" work itself with `ContinueWith`, rather than having to use yet another fresh async operation as in Example 2(a), lets us avoid both of the problems we just listed: We don't have to tie up an extra thread just to tack on some extra work to be done when the "future" is ready, and we don't have to perform a wakeup and context switch because the continuation can immediately run on the thread that fills the "future." For example:

```
// Example 2(b): Calling code, same as 2(a) except using
// ContinueWith method (if available)
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
    shared_future<bool> result;
    result = backgrounder.Save( filename );
    // queue up a continuation to notify ourselves of the
    // result once it's available
    result.ContinueWith( [=]
      SendSaveComplete( result->get() );
    } );
  }
  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

Prefer to use a `ContinueWith` style if it is available in your "futures library."

Option 3: Accept an Event or Callback (to "Push" to Caller)

Both of the alternatives we've just seen let the callee return a "future," which by default delivers a "pull" notification the caller can wait for. So far, we've left it to the caller to turn that "future" into a "push" notification (event or message) if it wants to be proactively notified when the result is available.

What if we want our callee to always offer proactive "push" notifications? The most general way to do that is to accept a callback to be invoked when the result is available:

```
// Example 3: Return value, using a callback
class Backgrounder {
public:
  void Save(
    string filename,
    function<void(bool)> returnCallback
  ) {
    a.Send( [=] {
      // … do the saving work …
      returnCallback( didItSucceed() ? true : false );
    } ); }
```

This is especially useful if the caller is itself an active object and gives a callback that is one of its own (asynchronous) methods. For example, this might be used from a GUI thread as follows:

```
class MyGUI {
public:
  // …

  // When the user clicks [Save]
  void OnSaveClick() {
    // …
    // … turn on saving icon, etc. …
    // …
```

```
    // pass a continuation to be called to give
    // us the result once it's available
    shared_future<bool> result;
    result = backgrounder.Save( filename,
       [=] { SendSaveComplete( result->get() ); } );
  }

  void OnSaveComplete( bool returnedValue ) {
    // … turn off saving icon, etc. …
  }
```

Since Example 3 uses a callback, it's worth mentioning a drawback common to all callback styles, namely: The callback runs on the callee's thread. In the aforementioned code that's not a problem because all the callback does is launch an asynchronous message event that gets queued up for the caller. But remember, it's always a good idea to do as little work as possible in the callee, and just firing off an asynchronous method call and immediately returning is a good practice for callbacks.

Getting Multiple or Interim Results

All of the aforementioned options deal well with return values and output parameters. Finally, however, what if we want to get multiple notifications before the final results, such as partial computation results, updated status such as progress updates, and so on?

We have two main options:

- Provide an explicit message queue or channel back to the caller, which can enqueue multiple results.
- Accept a callback to invoke repeatedly to pass multiple results back to the caller.

Example 4 will again use the callback approach. If the caller is itself an active object and the callback it provides is one of its own (asynchronous) methods, we've really combined the two paths and done both bullets at the same time. (Note: Here we're focusing on the interim progress via the `statusCallback`; for the return value, we'll again just use a "future" as in Examples 1 and 2.)

```
// Example 4: Returning partial results/status
class Backgrounder {
public:
  // Print() puts print result into spooler, returns one of:
  //   Error (failed, couldn't process or send to spooler)
  //   Printing (sent to spooler and already actively printing)
  //   Queued (sent to spooler but not yet actively printing)
  future<PrintStatus>
  Print(
    Data& data,
    function<void(PrintInfo)> statusCallback
  ) {
    auto p = make_shared<promise<PrintStatus>>();
    future<PrintStatus> ret = p->get_future();
    a.Send( [=, &data] {
      PrintInfo info;
      while( /* not done formatting the data */ ) {
        info.SetPercentDone( /*…*/ );
        statusCallBack( info );      // interim status
        // … do the printing work for another piece of the data …
      } while( /* not done formatting the data */ );
      p->set_value( /* … */ );        // set final result
      info.SetPercentDone( 100 );
      statusCallBack( info );          // final interim status
    } );
    return ret;
  }
```

This might be used in the GUI thread example as follows:

```
class MyGUI {
public:
  // …

  // When the user clicks [Print]
  void OnPrintClick() {
    // …
    // … turn on printing icon, etc. …
    // …
    // pass a continuation to be called to give
    // us the result once it's available
    shared_future<PrintStatus> result;
    result = backgrounder.Print( theData,
      < [=]( PrintInfo pi ) { SendPrintInfo( pi, result ); } );
  }

  void OnPrintInfo(
    PrintInfo pi,
    shared_future<PrintStatus> result
  ) {
    // … update print progress bar to
    //     pi.GetPercentDone(), etc. …
    // if this is the last notification
    // (100% done, or result is ready)
    if( result.is_ready() ) {
      // … turn off printing icon, etc. …
    }
  }
```

Summary

To express return values and "out" parameters from an asynchronous function, including an active object method, either:

- Return a "future" to invoke that the caller can "pull" the result from (Example 1) or convert it to a "push" (Examples 2(a) and 2(b), and prefer to use `ContinueWith` where available); or
- Accept a callback to invoke to "push" the result to the caller when ready (Example 3).

To return multiple partial results, such as partial computations or even just "percent done" progress information, also use a callback (Example 4).

Whenever you provide callbacks, remember that they are running in the callee's context, so we want to keep them as short and noninvasive as possible. One good practice is to have the callback just fire off asynchronous messages or method calls and return immediately.

On Deck

Besides moving work off to a background thread, what else could we use an active object for? We'll consider an example next time, but for now, here's a question for you to ponder: How might you use an active object to replace a mutex on some shared state? Think about ways you might approach that problem, and we'll consider an example in my next column.

References

[1] H. Sutter. Prefer Using Active Objects Instead of Naked Threads. Dr. Dobb's Digest, June 2010.

[2] H. Sutter. Prefer Futures to Baked-In 'Async APIs'. Dr. Dobb's Digest, January 2010.

[3] Task.ContinueWith Method (Action<Task>); MSDN.

Herb Sutter is a bestselling author and consultant on software development topics, and a software architect at Microsoft. He can be contacted at www.gotw.ca.