Subject: About storing references and pointers to callbacks.
Posted by Oblivion on Sun, 25 Jun 2017 19:13:38 GMT
View Forum Message <> Reply to Message

Hello,

I'd like to ask you a question. Here is the problem I need to solve:

I need to store pointers of complex objects, say, Streams, to callbacks (e.g. for deferred/async file reads and writes) so that I can access them only when I need them.
The culprit is that I don't want the caller function (or callback) to own those objects. Knowing their current state -whether they are destroyed or existing- to proceed or to halt is sufficient.

I know that simply passing pointers is dangerous, since the life time of objects can vary and not be strictly determined especially on complex applications.
Now, I know C++11 and above versions of C++ standard have std::shared_ptr and std::weak_ptr suitable for this purpose.
Also U++ has something similar: Ptr and Pte.

What would be the U++ way to handle these situations?

```
MyClass::MyAsyncReadMethod(Stream& s)
{
	Stream *p = &s; // This is a bad, very bad practice!

	vector_containing_callbacks.Add() << [=] {

			p->GetLine() // !! might eat cats!

			    ?? // How should I proceed?
			       // Should I use shared_ptr && weak_ptr?
			       // Or A Stream derivative with Ptr && Pte?
			       // or another alternative?
	};
}
```

Any suggestions or ideas will be much appreciated. Thanks.

Bestt regards,
Oblivion

Subject: Re: About storing references and pointers to callbacks.

Posted by mirek on Sun, 25 Jun 2017 19:34:16 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Sun, 25 June 2017 21:13Hello,

I'd like to ask you a question. Here is the problem I need to solve:

I need to store pointers of complex objects, say, Streams, to callbacks (e.g. for deferred/async file reads and writes) so that I can access them only when I need them.
The culprit is that I don't want the caller function (or callback) to own those objects. Knowing their current state -whether they are destroyed or existing- to proceed or to halt is sufficient.

I know that simply passing pointers is dangerous, since the life time of objects can vary and not be strictly determined especially on complex applications.
Now, I know C++11 and above versions of C++ standard have std::shared_ptr and std::weak_ptr suitable for this purpose.
Also U++ has something similar: Ptr and Pte.

What would be the U++ way to handle these situations?

```
MyClass::MyAsyncReadMethod(Stream& s)
{
	Stream *p = &s; // This is a bad, very bad practice!

	vector_containing_callbacks.Add() << [=] {

			p->GetLine() // !! might eat cats!

			  ?? // How should I proceed?
			    // Should I use shared_ptr && weak_ptr?
			    // Or A Stream derivative with Ptr && Pte?
			    // or another alternative?
	};
}
```

Any suggestions or ideas will be much appreciated. Thanks.

Bestt regards,
Oblivion

It all depends on context, which you do not provide. However, in general, I would guess in similar situations destroying stream before MyClass does not make much sense from user perspective. So you can go along with it, even maybe with [=, &s] and just put something like "Stream must exist through the duration of MyClass (or until calling some methods that removes it from MyClass).

Subject: Re: About storing references and pointers to callbacks.
Posted by mirek on Sun, 25 Jun 2017 19:36:01 GMT
View Forum Message <> Reply to Message

Or, reading the name of method, maybe it rather needs signature like

MyClass::MyAsyncReadMethod(Function<String> out)

?

Subject: Re: About storing references and pointers to callbacks.
Posted by Oblivion on Sun, 25 Jun 2017 19:43:30 GMT
View Forum Message <> Reply to Message

Thank you for your quick reply!

Here is the actual code:

```
void SFtp::StartGet(Stream& out)
{
 packet_length = 0;
 Stream *io_stream = &out;
 AddJob() << [=] {
  Buffer<char> buffer(chunk_size);
  int rc = libssh2_sftp_read(handle, buffer, chunk_size);
  if(rc > 0) {
   io_stream->Put(buffer, rc);
   if(WhenRead(dir_entry.GetSize(), io_stream->GetSize()))
    Error(-1, t_("File download aborted."));
   return true;
  }
  else
  if(rc == 0) {
   LLOG(Format("++ SFTP: %ld of %ld bytes successfully read.", io_stream->GetSize(),
dir_entry.GetSize()));
   return false;
  }
  if(!WouldBlock())
   Error();
  return true;
 };
}
```

This method is called by the user who wants to download a file from the server. It should be called before proceeding of a "job queue" (replicates the HttpRequest class' async interface but with

callbacks).
This method stores a pointer to callback which is queued in a Vector<Function<bool(void)>>.

```
SFtp sftpclient;

FileOut file1, file2, file3;


sftpclient.StartGet(file1);
sftpclient.StartGet(file2);
sftpclient.StartGet(file3);


//...
// Then below,
// Something along these lines:
while(1) {
 sftpclient.Do();
 if(sftpClient.InProgress())
  // do something
 else
  // do something else..

}
```

Subject: Re: About storing references and pointers to callbacks.
Posted by mirek on Mon, 26 Jun 2017 07:26:41 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Sun, 25 June 2017 21:43Thank you for your quick reply!

Here is the actual code:

```
void SFtp::StartGet(Stream& out)
{
 packet_length = 0;
 Stream *io_stream = &out;
 AddJob() << [=] {
  Buffer<char> buffer(chunk_size);
  int rc = libssh2_sftp_read(handle, buffer, chunk_size);
  if(rc > 0) {
   io_stream->Put(buffer, rc);
   if(WhenRead(dir_entry.GetSize(), io_stream->GetSize()))
    Error(-1, t_("File download aborted."));
   return true;
  }
  else
  if(rc == 0) {
   LLOG(Format("++ SFTP: %ld of %ld bytes successfully read.", io_stream->GetSize(),
dir_entry.GetSize()));
   return false;
  }
  if(!WouldBlock())
   Error();
  return true;
 };
}
```

This method is called by the user who wants to download a file from the server. It should be called before proceeding of a "job queue" (replicates the HttpRequest class' async interface but with callbacks).
This method stores a pointer to callback which is queued in a Vector<Function<bool(void)>>.

```
SFtp sftpclient;

FileOut file1, file2, file3;


sftpclient.StartGet(file1);
sftpclient.StartGet(file2);
sftpclient.StartGet(file3);


//...
// Then below,
// Something along these lines:
while(1) {
 sftpclient.Do();
 if(sftpClient.InProgress())
  // do something
 else
  // do something else..

}
```

I am confused. What are these 3 FileOut files in the example? What is supposed to go to them? 3 separate files? (but then I would expect some url as partof startget). Or parts of single file? That would be bad...

Other than that, the lowest level you have is "buffer" and "chunk_size". I think that lowest level of sftp should reflect that.

e.g.

void SFtp::StartGet(Event<void *ptr, int size>)

Subject: Re: About storing references and pointers to callbacks.
Posted by Oblivion on Mon, 26 Jun 2017 10:14:52 GMT

View Forum Message <> Reply to Message

Hello Mirek,

Sorry for the confusing code. That was not very informative, I admit.
They are not parts of a single file, they are separate, queued downloads.

Sftp class has both low level commands (corresponding to fopen, fstat, fread, fwrite) and their high level counterparts.
Below code simply demonstrates the low level code.

A real downloader with low level commands is like this: (usually we won't use these low level commands)

```
#include <Core/Core.h>
#include <SSH/SSH.h>

using namespace Upp;



CONSOLE_APP_MAIN
{
 const char *remote_path = "readme.txt";

 Ssh ssh;
 Cout() << "Connecting to Rebex public sftp test server...\n";

 // Let us connect in a blocking way.
 bool b = ssh.Connect("test.rebex.net", 22, "demo", "password");
 if(b) {
  // Note that below code respresents low level, async sftp file transfer.

       FileOut local_file("readme_downloaded.txt");

  // SFtp file attributes are stored in this structure, and used by low-level methods.
     SFtpAttrs file_attrs;

  // Now let us request a SFtp channel.
  SFtp sftp(ssh);

  // Chunk size is used only in gets and puts. Default chunk size is 32K
  //sftp.ChunkSize(16 * 1024);

  // Now let us queue the Sftp commands.
  sftp.StartInit();

   // Asynhronous calls can be queued. And they have both high level and low level
   // interface.
   // For example: StartGet has two variants:
   // 1) A low levet StartGet(): Requires a file handle (fopen) and file info (fstat) to operate on.
   //    So It requires StartOpen() and StartGetStat() methods to be called first.
   //    For this purpose, we can queue them and later run them:
                        // (Note that, any error while processing the commands will halt the queue, and
clean-up the mess. So it's safe.)

      sftp.StartOpen(remote_path, SFtp::READ, 0755);
     sftp.StartGetStat(file_attrs);
         sftp.StartGet(local_file);
         sftp.StartClose();

   // 2) A high level StartGet(): This method combines the above StartOpen, StartGetStat,
   // StartGet, and StartClose respectively, and handles errors internally:
   //
   // sftp.StartGet(local_file, remote_path, SFtp::READ, 0755);

  sftp.StartStop();

  // Now that we have queued up the download, using low level api, we can run the queue
asynchronously.
  // Note that we are running a single Sftp instance here.
  // We can do this with multiple instances (using an Array<SFtp>) and multiple commands too.

  while(sftp.Do()) {
   if(!sftp.InProgress()) {
    if(sftp.IsSuccess())
     Cout() << "File successfully downloaded.\n";
    else
     Cout() << "File download failed. Reason: " << sftp.GetErrorDesc() << "\n";
    break;
   }
  }
 }
 else
  Cout() << ssh.GetErrorDesc() << "\n";
 ssh.Disconnect();
}
```

The problem can arise when the loop is called elsewhere (where it may outlive the FileOut streams.). I guess it is simply enough to warn the users about this in the docs.

Subject: Re: About storing references and pointers to callbacks.
Posted by mirek on Mon, 26 Jun 2017 11:11:16 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Mon, 26 June 2017 12:14The problem can arise when the loop is called elsewhere (where it may outlive the FileOut streams.). I guess it is simply enough to warn the users about this in the docs.

Well, I have some reservation about the whole api design, but yes, this case I would say it is ok and has to be documented.

Subject: Re: About storing references and pointers to callbacks.
Posted by Oblivion on Mon, 26 Jun 2017 11:16:38 GMT
View Forum Message <> Reply to Message

Quote: Well, I have some reservation about the whole api design, but yes, this case I would say it is ok and has to be documented.

Thank you Mirek.

I hope I'm not taking much of your time, but could you share your thoughts on that?
So that I can revise the code before publishing it.

Subject: Re: About storing references and pointers to callbacks.
Posted by mirek on Wed, 05 Jul 2017 12:02:46 GMT

View Forum Message <> Reply to Message

Oblivion wrote on Mon, 26 June 2017 13:16Quote: Well, I have some reservation about the whole api design, but yes, this case I would say it is ok and has to be documented.

Thank you Mirek.

I hope I'm not taking much of your time, but could you share your thoughts on that?
So that I can revise the code before publishing it.

Well, _right now_ I feel a bit uneasy about the whole asynchronous operations queue. Now, it is likely that the whole concept is forced by the nature of problem, but I would rather liked to have some "single request" tool.

Actually, I think that your original question suggest exatly that long queues that store references to external objects are troublesome...

Also, I think that there always should be Event<void *, int> basic variant for data outputs.

Subject: Re: About storing references and pointers to callbacks.
Posted by Oblivion on Wed, 05 Jul 2017 21:01:11 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 05 July 2017 15:02Oblivion wrote on Mon, 26 June 2017 13:16Quote:
Well, I have some reservation about the whole api design, but yes, this case I would say it is ok and has to be documented.

Thank you Mirek.

I hope I'm not taking much of your time, but could you share your thoughts on that?
So that I can revise the code before publishing it.

Well, _right now_ I feel a bit uneasy about the whole asynchronous operations queue. Now, it is likely that the whole concept is forced by the nature of problem, but I would rather liked to have some "single request" tool.

Actually, I think that your original question suggest exatly that long queues that store references to external objects are troublesome...

Also, I think that there always should be Event<void *, int> basic variant for data outputs.

Hello Mirek,

Thanks for the answer.

I understand your uneasiness about the queue. But it allows writing "single request" tools too.
I mean it can be easily programmed to process only one asynchronous or synchronous job at a time.
For example, it all takes adding a single line to the StartGet() method above to make it a "single request" (adding JobQueue::ClearQueue() will clear the queue if there are pending jobs prior to current call). However, as you might have already noticed in the SSH package I published (which is still a work-in-progress), I did not walk that path. I simply provided non-blocking calls and their blocking counterparts. And that's intentional. My experience with the asynchronous sockets in the past years have taught me that performing a single operation with them is trivial but "a set of sequential operations" is at best tedious. Whole point of the queue is to simplify that process for the "user" (developer) and make the code readable by reducing complexity. It proved well up to the task. (E.g. I have several unpublished packages such as a dbus client which uses the JobQueue, and they all work fine with several different scenarios. The queue model simplified things a lot in the asynchronous sockets' realm.).

As for the data outputs using an Event<void*, int>, indeed, that's a valid point too. I should think on it.

Best regards,
Oblivion