Programming with curlpp

Jean-Philippe Barrette-LaPierre

July 14, 2004

1 About this Document

This document attempts to describe the general principles and some basic approaches to consider when programming with curlpp. Don't forget that curlpp is a C++wrapper of libcurl, so curlpp need libcurl to be allready installed.

This document will refer to 'the user' as the person writing the source code that uses curlpp. That would probably be you or someone in your position. What will be generally refered to as 'the program' will be the collected source code that you write that is using curlpp for transfers. The program is outside curlpp and curlpp is outside of the program.

To get the more details on all options and functions described herein, please refer to their respective man pages.

2 Building

There are many different ways to build C++ programs. This chapter will assume a unix-style build process. If you use a different build system, you can still read this to get general information that may apply to your environment as well. Note that curlpp need libcurl to be allready installed.

2.1 Compiling the Program

Your compiler needs to know where the curlpp and libcurl headers are located. Therefore you must set your compiler's include path to point to the directory where you installed them. The 'curlpp-config'[1] tool can be used to get this information:

```
$ curlpp-config --cflags
```

[1]The curlpp-config tool, who wrap all fonctions of curl-config, is generated at build-time (on unix-like systems) and should be installed with the 'make install' or similar instruction that installs the library, header files, man pages etc.

2.2 Linking the Program with curlpp

When having compiled the program, you need to link your object files to create a single executable. For that to succeed, you need to link with curlpp and possibly also with other libraries that curlpp itself depends on (like libcurl). Like OpenSSL libraries, but even some standard OS libraries may be needed on the command line. To figure out which flags to use, once again the 'curlpp-config' tool comes to the rescue:

```
$ curlpp-config --libs
```

2.3 SSL or Not

curlpp, in fact libcurl, can be built and customized in many ways. One of the things that varies from different libraries and builds is the support for SSL-based transfers, like HTTPS and FTPS. If OpenSSL was detected properly by libcurl at build-time, curlpp will be built with SSL support. To figure out if an installed curlpp has been built with SSL support enabled, use 'curlpp-config' like this:

```
$ curlpp-config --feature
```

And if SSL is supported, the keyword 'SSL' will be written to stdout, possibly together with a few other features that can be on and off on different curlpps.

2.4 Portable Code in a Portable World

The people behind libcurl have put a considerable effort to make libcurl work on a large amount of different operating systems and environments.

You program curlpp the same way on all platforms that curlpp runs on. There are only very few minor considerations that differs. If you just make sure to write your code portable enough, you may very well create yourself a very portable program. curlpp shouldn't stop you from that.

3 Global Preparation

The program must initialize some of the curlpp functionality globally. That means it should be done exactly once, no matter how many times you intend to use the library. Once for your program's entire life time. This is done using

```
curlpp::initialize( long flags = CURL_GLOBAL_ALL )
```

and it takes one parameter which is a bit pattern that tells curlpp what to intialize.

1. `CURL_GLOBAL_ALL` will make it initialize all known internal sub modules, and might be a good default option. The current two bits that are specified are:

2. `CURL_GLOBAL_WIN32` which only does anything on Windows machines. When used on a Windows machine, it'll make curlpp intialize the win32 socket stuff. Without having that initialized properly, your program cannot use sockets properly. You should only do this once for each application, so if your program already does this or of another library in use does it, you should not tell curlpp to do this as well.

3. `CURL_GLOBAL_SSL` which only does anything on curlpps compiled and built SSL-enabled. On these systems, this will make curlpp init OpenSSL properly for this application. This is only needed to do once for each application so if your program or another library already does this, this bit should not be needed.

curlpp has a default protection mechanism that detects if `curlpp::initialize()` hasn't been called by the time `curlpp::easy::easy()` or `curlpp::easy::perform()` is called and if that is the case, curlpp throws a `curlpp::logic_error`.

When the program no longer uses curlpp, it should call `curlpp::terminate()`, which is the opposite of the init call. It will then do the reversed operations to cleanup the resources the `curlpp::initialize()` call initialized.

Repeated calls to `curlpp::initialize()` and `curlpp::terminate()` must not be made. They must only be called once each.

4 Handle the Easy curlpp

To use the easy interface, you must first create yourself an easy handle. You need one handle for each easy session you want to perform. Basicly, you should use one handle for every thread you plan to use for transferring. You must never share the same handle in multiple threads.

Get an easy handle with

```
curlpp::easy easyhandle;
```

It creates an easy handle. Using that you proceed to the next step: setting up your preferred actions. A handle is just a logic entity for the upcoming transfer or series of transfers. Use `curlpp::http_easy` and `curlpp::ftp_easy` for options specific to, respectively, HTTP or FTP requests.

You set properties and options for this handle using member functions, or their registry, we will discuss about registry later. They control how the subsequent transfer or transfers will be made. Options remain set in the handle until set again to something different. Alas, multiple requests using the same handle will use the same options.

Many of the informationals you set in curlpp are "strings", pointers to data terminated with a zero byte. Keep in mind that when you set strings with member functions, curlpp will copy the data. It will not merely point to the data. You don't need to make sure that the data remains available for curlpp.

One of the most basic properties to set in the handle is the URL. You set your preferred URL to transfer with `void curlpp::easy::url(const char *link)` in a manner similar to:

```
easyhandle.url("http://curl.haxx.se/");
```

Let's assume for a while that you want to receive data as the URL indentifies a remote resource you want to get here. Since you write a sort of application that needs this transfer, I assume that you would like to get the data passed to you directly instead of simply getting it passed to stdout. So, you write your own class that devive from `curlpp::output_trait` and reimplement:

```
virtual size_t curlpp::ouput_trait::write( void *buffer, size_t length )
```

You tell curlpp to pass all data to this class by issuing a function similar to this:

```
devived_trait_class my_trait;
easyhandle.m_ouput_trait.trait( &my_trait );
```

But you're not obliged to derive a class if you just want to put the buffer in a file. We give you some basic trait class that you can use. there's `curlpp::ostream_trait` or `curlpp::ofile_trait`, for a `std::ostream` or a `FILE*`, respectively. You can use a `curlpp::ostream_trait` in a similar way to:

```
std::ofstream my_file( "./my_file.txt" );
easyhandle.m_body_storage.data( &my_file );
```

But if you need to do a special treatment to your buffer you'll need to derive from the `curlpp::easy::output_trait`. The `curlpp::storage<T>::data` permit you to bind a particular type to a trait. It gives you the possibility, as the example before, to map a `std::ofstream` to a `curlpp::ostream_trait`, without requiring you to create an instance of the latter. Note that curlpp is creating this instance internally, the only advantage is that it save you from writting one another line of code. If you want to be able to use the `curlpp::storage<T>::data` function with your own trait, you will need to specialize the `curlpp::trait_resolver` function(check `trait_resolver.hpp` for more details).

curlpp offers its own default internal class that'll take care of the data if you don't set a trait. It will then simply output the received data to stdout. The default trait for `storage<curlpp::body_storage> curlpp::easy::m_body_storage` is `curlpp::stdout_trait` that is a `curlpp::ofile_trait` with `FILE *stdout` .

There are of course many more options you can set, and we'll get back to a few of them later. Let's instead continue to the actual transfer:

```
easyhandle.perform();
```

The `curlpp::easy::perform()` will connect to the remote site, do the necessary commands and receive the transfer. Whenever it receives data, it calls the trait function we previously set. The function may get one byte at a time, or it may get many kilobytes at once. curlpp delivers as much as possible as often as possible. Your trait function should return the number of bytes it "took care of". If that is not the exact same amount of bytes that was passed to it, curlpp will abort the operation and throw an exception.

When the transfer is complete, the function throws a `curlpp::exception` if it doesn't succeeded in its mission. the `const char *curlpp::exception::what()` will return the human readable reason of failure.

If you then want to transfer another file, the handle is ready to be used again. Mind you, it is even preferred that you re-use an existing handle if you intend to make another transfer. curlpp will then attempt to re-use the previous connections.

5 Multi-threading issues

curlpp is completely thread safe, except for two issues: signals and alarm handlers. Signals are needed for a SIGPIPE[2] handler, and the alarm() syscall is used to catch timeouts (mostly during DNS lookup).

So when using multiple threads you should first ignore SIGPIPE in your main thread and set the `curlpp::easy::signal( bool disactivate_signals = false )` option to `false` for all handles.

Everything will work fine except that timeouts are not honored during the DNS lookup - this would require some sort of asynchronous DNS lookup (which is planned for a future curlpp version).

6 Exceptions issues

As previously said, curlpp (libcurl in fact) offer the possibility te reimplement the data writing/reading functions. Those functions called from within libcurl might raise exceptions. Raising an exception in C code, might cause problems. curlpp protect you from doing so. All exceptions are catched by curlpp before it could cause damages to libcurl. If you want to raise an exception within traits, you need do this:

```
MyException *myException = new MyException("Exception Raised");
curlpp::raiseException(myException);
```

Then, the `curlpp::easy::perform()` will raise your exception. If an exception is raised but not by this mechanism, a `curlpp::unknown_exception_error` will be raised.

[2]For SIGPIPE info see the UNIX Socket FAQ at http://www.unixguide.net/network/socketfaq/2.22.shtml

7 When It Doesn't Work

There will always be times when the transfer fails for some reason. You might have set the wrong curlpp option or misunderstood what the curlpp option actually does, or the remote server might return non-standard replies that confuse the library which then confuses your program.

There's one golden rule when these things occur: set the `curlpp::easy::verbose(bool activate)` option to `true`. It'll cause the library to spew out the entire protocol details it sends, some internal info and some received protcol data as well (especially when using FTP). If you're using HTTP, adding the headers in the received output to study is also a clever way to get a better understanding what the server behaves the way it does. Include headers in the normal body output with `curlpp::easy::headers(bool activate = true)` set to `true`.

Of course there are bugs left. We need to get to know about them to be able to fix them, so we're quite dependent on your bug reports! When you do report suspected bugs in curlpp, please include as much details you possibly can: a protocol dump that `curlpp::easy::verbose(bool activate)` produces, library version, as much as possible of your code that uses curlpp, operating system name and version, compiler name and version etc.

Getting some in-depth knowledge about the protocols involved is never wrong, and if you're trying to do funny things, you might very well understand curlpp and how to use it better if you study the appropriate RFC documents at least briefly.

Upload Data to a Remote Site

curlpp tries to keep a protocol independent approach to most transfers, thus uploading to a remote FTP site is very similar to uploading data to a HTTP server with a PUT request.

Of course, first you either create an easy handle or you re-use one existing one. Then you set the URL to operate on just like before. This is the remote URL, that we now will upload.

Since we write an application, we most likely want curlpp to get the upload data by asking us for it. To make it do that, we set the read trait class and You must derive from `curlpp::input_trait` and reimplement:

```
virtual size_t curlpp::input_trait::read( void *buffer, size_t length )
```

Where buffer is the pointer to data and length is the size of the buffer and therefore also the maximum amount of data we can return to curlpp in this call.

```
devived_trait_class my_trait;
easyhandle.m_input_trait.trait( &my_trait );
```

Tell curlpp that we want to upload:

```
curlpp::easy::upload( true );
```

Like we say earlier, you're not obliged to derive a class if you just want to put in the buffer is a file. We give you some basic trait class that you can use. there's `curlpp::istream_trait` or `curlpp::ifile_trait`, for a `std::istream` or a `FILE*`, respectively. You can use a `curlpp::istream_trait` in a similar way to:

```
std::ifstream my_file( "./my_file.txt" );
curlpp::istream_trait my_trait( &my_file );
easyhandle.m_input_storage.trait( &my_trait );
```

A few protocols won't behave properly when uploads are done without any prior knowledge of the expected file size. HTTP PUT is one example[3]. So, set the upload file size using the `curlpp::easy::infile_size( long size )` like this:

```
easyhandle.infile_size( file_size );
```

When you call `curlpp::easy::perform()` this time, it'll perform all the necessary operations and when it has invoked the upload it'll call your supplied callback to get the data to upload. The program should return as much data as possible in every invoke, as that is likely to make the upload perform as fast as possible. The callback should return the number of bytes it wrote in the buffer. Returning 0 will signal the end of the upload.

8 Passwords

Many protocols use or even require that user name and password are provided to be able to download or upload the data of your choice. libcurl offers several ways to specify them.

Most protocols support that you specify the name and password in the URL itself. libcurl will detect this and use them accordingly. This is written like this:

protocol://user:password@example.com/path/

If you need any odd letters in your user name or password, you should enter them URL encoded, as %XX where XX is a two-digit hexadecimal number.

libcurl also provides options to set various passwords. The user name and password as shown embedded in the URL can instead get set with the `curlpp::easy::user_pwd( const st` function. The argument passed to libcurl should be a std::string to a string in the format "user:password:". In a manner like this:

```
easyhandle.user_pwd( "myname:thesecret" );
```

Another case where name and password might be needed at times, is for those users who need to athenticate themselves to a proxy they use. libcurl offers another fcuntion for this, the `curlpp::easy::proxy_user_pwd( const std::string & )`

[3]HTTP PUT without knowing the size prior to transfer is indeed possible, but curlpp does not support the chunked transfers on uploading that is necessary for this feature to work. We'd gratefully appreciate patches that bring this functionality...

function. It is used quite similar to the `curlpp::easy::proxy_user_pwd( const std::string & )` function like this:

```
easyhandle.proxy_user_pwd( "myname:thesecret" );
```

There's a long time unix "standard" way of storing ftp user names and passwords, namely in the `\$HOME/.netrc` file. The file should be made private so that only the user may read it (see also the "Security Considerations" chapter), as it might contain the password in plain text. libcurl has the ability to use this file to figure out what set of user name and password to use for a particular host. As an extension to the normal functionality, libcurl also supports this file for non-FTP protocols such as HTTP. To make curl use this file, use the `curlpp::easy::netrc( curlpp::netrc::netrc_option )` function:

```
curlpp::easy::netrc( curlpp::netrc::required );
```

And a very basic example of how such a .netrc file may look like:
machine myhost.mydomain.com login userlogin password secretword

All these examples have been cases where the password has been optional, or at least you could leave it out and have libcurl attempt to do its job without it. There are times when the password isn't optional, like when you're using an SSL private key for secure transfers.

You can in this situation either pass a password to libcurl to use to unlock the private key, or you can let libcurl prompt the user for it. If you prefer to ask the user, then you can provide your own callback function that will be called when libcurl wants the password. That way, you can control how the question will appear to the user.

To pass the known private key password to libcurl:

```
curlpp::easy::ssl_key_passwd( "keypassword" );
```

To make a password callback:

```
devived_passwd_trait_class my_trait;
easyhandle.m_passwd_trait.trait( &my_trait );
```