Using libcurl on OSX

Posted on March 21, 2012 by admin

getting started with libcurl

As part of on-going development of our Edge platform, our web crawler needs to repeatedly download web pages of content on multiple threads. Oddly, when looking around, there are remarkably few packages and libraries for C++ to handle HTTP. HTTP is not overly complex and its a simple enough matter to use a framework like BOOST::ASIO for example and code it yourself, but really, we just need a simple class that will get a URL for us.

After an initial sweep, we settled on two main contenders; **cpp-netlib** and **libcurl**. They represent very different approaches to the task. cpp-netlib is/was being developed as a potential candidate for inclusion in BOOST and uses a nice modern template metaprogramming model.

I had a good play with cpp-netlib and got it doing most of what I wanted. It requires BOOST which is not a big concern for us as we are using a fair amount of the BOOST libraries anyway, although if you are not, you will need to get BOOST installed (easy enough using macports on OSX or indeed with a download and build). It also has its own build procedure that is somewhat at odds with the traditional BOOST build methods for BOOST candidate libraries.

In the end, for me, cpp-netlib was missing one or two specific features that we need and was perhaps a little too new and evolving. Its also designed as an abstraction to support many network protocols – not just HTTP, which might perhap makes it a little complex for those who simply want to download some web content. The team is very responsive and its very much a live project right now. It certainly something we will be keeping an eye on and re-visiting from time to time. If you need an HTTP server, its going to be a fine choice – If you just want to get some content down, others might be easier.

Libcurl is well-established (for well-established, read 'old as the hills'). There is a ton of information out there and the code is nice and stable, supported with bindings for every possible language and also actively being developed. It underlies the 'curl' tool that is available on most systems (it's installed by default on OSX). One issue is that for C++ we are talking about an old-school 'C' style API. It uses a model of callbacks to gather the content and header information that don't sit too nicely with a class model in C++ – it's certainly at the other end of the modernity spectrum compared with cpp-netlib.

While there is a wealth of info out there on libcurl, there is not too much concise getting started information including installing and building your first app to do the most common thing – get a url's content and headers in a nice, simple class.

/usr/local/include for headers and /usr/local/lib for libraries. If your installation isn't, then you can set 'Header search paths' and 'Library search paths' in the 'Search paths' section of the Build Settings.

You will however need to add the library to your project so the linker knows to link against it. in the Build Settings, in the 'Linking' section, add -lcurl to the 'other linker options' item.

curly.h:

```
#ifndef curly_h
#define curly_h

#include <string>
#include <vector>
#include <curl/curl.h>

class Curly{
private:
    std::string        mContent;
    std::string        mType;
    std::vector        mHeaders;
```

```
    unsigned int       mHttpStatus;
    CURL*              pCurlHandle;
    static size_t      HttpContent(void* ptr, size_t size,
                        size_t nmemb, void* stream);
    static size_t      HttpHeader(void* ptr, size_t size,
                        size_t nmemb, void* stream);

public:
    Curly():pCurlHandle(curl_easy_init()){};  // constructor
    ~Curly(){};
    CURLcode    Fetch (std::string);

    inline std::string   Content()    const { return mContent; }
    inline std::string   Type()       const { return mType; }
    inline unsigned int  HttpStatus() const { return mHttpStatus; }
    inline std::vector   Headers()    const { return mHeaders; }
};

#endif
```

curly.cpp

```
#include "curlget.h"

CURLcode Curly::Fetch(std::string url){

    // clear things ready for our 'fetch'
    mHttpStatus = 0;
    mContent.clear();
    mHeaders.clear();

    // set our callbacks
    curl_easy_setopt(pCurlHandle , CURLOPT_WRITEFUNCTION, HttpContent);
    curl_easy_setopt(pCurlHandle, CURLOPT_HEADERFUNCTION, HttpHeader);
    curl_easy_setopt(pCurlHandle, CURLOPT_WRITEDATA, this);
    curl_easy_setopt(pCurlHandle, CURLOPT_WRITEHEADER, this);

    // set the URL we want
    curl_easy_setopt(pCurlHandle, CURLOPT_URL, url.c_str());

    //  go get 'em, tiger
    CURLcode curlErr = curl_easy_perform(pCurlHandle);
    if (curlErr == CURLE_OK){

        // assuming everything is ok, get the content type and status code
        char* content_type = NULL;
        if ((curl_easy_getinfo(pCurlHandle, CURLINFO_CONTENT_TYPE,
              &content_type)) == CURLE_OK)
            mType = std::string(content_type);

        unsigned int http_code = 0;
        if((curl_easy_getinfo (pCurlHandle, CURLINFO_RESPONSE_CODE,
              &http_code)) == CURLE_OK)
            mHttpStatus = http_code;

    }
    return curlErr;
}

size_t Curly::HttpContent(void* ptr, size_t size,
                          size_t nmemb, void* stream) {

        Curly* handle = (Curly*)stream;
        size_t data_size = size*nmemb;
    if (handle != NULL){
        handle->mContent.append((char *)ptr,data_size);
```

/usr/local/include for headers and /usr/local/lib for libraries. If your installation isn't, then you can set 'Header search paths' and 'Library search paths' in the 'Search paths' section of the Build Settings.

You will however need to add the library to your project so the linker knows to link against it. in the Build Settings, in the 'Linking' section, add -lcurl to the 'other linker options' item.

curly.h:

```
#ifndef curly_h
#define curly_h

#include <string>
#include <vector>
#include <curl/curl.h>

class Curly{
private:
    std::string        mContent;
    std::string        mType;
    std::vector        mHeaders;
```

```
    unsigned int       mHttpStatus;
    CURL*              pCurlHandle;
    static size_t      HttpContent(void* ptr, size_t size,
                        size_t nmemb, void* stream);
    static size_t      HttpHeader(void* ptr, size_t size,
                        size_t nmemb, void* stream);

public:
    Curly():pCurlHandle(curl_easy_init()){};  // constructor
    ~Curly(){};
    CURLcode    Fetch (std::string);

    inline std::string   Content()    const { return mContent; }
    inline std::string   Type()       const { return mType; }
    inline unsigned int  HttpStatus() const { return mHttpStatus; }
    inline std::vector   Headers()    const { return mHeaders; }
};

#endif
```

curly.cpp

```
#include "curlget.h"

CURLcode Curly::Fetch(std::string url){

    // clear things ready for our 'fetch'
    mHttpStatus = 0;
    mContent.clear();
    mHeaders.clear();

    // set our callbacks
    curl_easy_setopt(pCurlHandle , CURLOPT_WRITEFUNCTION, HttpContent);
    curl_easy_setopt(pCurlHandle, CURLOPT_HEADERFUNCTION, HttpHeader);
    curl_easy_setopt(pCurlHandle, CURLOPT_WRITEDATA, this);
    curl_easy_setopt(pCurlHandle, CURLOPT_WRITEHEADER, this);

    // set the URL we want
    curl_easy_setopt(pCurlHandle, CURLOPT_URL, url.c_str());

    //  go get 'em, tiger
    CURLcode curlErr = curl_easy_perform(pCurlHandle);
    if (curlErr == CURLE_OK){

        // assuming everything is ok, get the content type and status code
        char* content_type = NULL;
        if ((curl_easy_getinfo(pCurlHandle, CURLINFO_CONTENT_TYPE,
              &content_type)) == CURLE_OK)
            mType = std::string(content_type);

        unsigned int http_code = 0;
        if((curl_easy_getinfo (pCurlHandle, CURLINFO_RESPONSE_CODE,
              &http_code)) == CURLE_OK)
            mHttpStatus = http_code;

    }
    return curlErr;
}

size_t Curly::HttpContent(void* ptr, size_t size,
                          size_t nmemb, void* stream) {

        Curly* handle = (Curly*)stream;
        size_t data_size = size*nmemb;
    if (handle != NULL){
        handle->mContent.append((char *)ptr,data_size);
```

```
,

size_t Curly::HttpContent(void* ptr, size_t size,
                                  size_t nmemb, void* stream)

        Curly* handle = (Curly*)stream;
        size_t data_size = size*nmemb;
    if (handle != NULL){
        handle->mContent.append((char *)ptr,data_size);
    }
        return data_size;
}

size_t Curly::HttpHeader(void* ptr, size_t size,
                                  size_t nmemb, void* stream)

    Curly* handle = (Curly*)stream;
        size_t data_size = size*nmemb;
    if (handle != NULL){
        std::string header_line((char *)ptr,data_size);
        handle->mHeaders.push_back(header_line);
```

```
    }
    return data_size;
}
```

main.cpp

```
#include
#include "curly.h"

int main (int argc, const char * argv[])
{

    Curly curly;

    if (curly.Fetch("http://www.dahu.co.uk") == CURLE_OK

        std::cout << "status: " << curly.HttpStatus() <<
        std::cout << "type: " << curly.Type() << std::en
        std::vector headers = curly.Headers();

        for(std::vector::iterator it = headers.begin();
                it != headers.end(); it++)
            std::cout << "Header: " << (*it) << std::end

        std::cout << "Content:\n" << curly.Content() <<
    }

    return 0;
}
```

A brief explanation

The constructor for the ‘curly’ class calls

```
curl_easy_init()
```

to initialise curl, and importantly, the deconstructor calls

```
curl_easy_cleanup(handle)
```

to release the resources libcurl has used. Curl provides a wealth of calls to do pretty much everything you might ever want to do with HTTP, including handling redirects, cookies and setting custom headers and also provides a set of ‘easy’ calls to do the most common things with little or no fuss, which is what we are using in this example.

Once we have an instance created, we can call ‘Fetch’ and pass it a URL to get. Note again that we are not providing any parsing or validation of the URL for this simple example – you will need to ensure that yourself before you call Fetch.

In Fetch, first we clean down our members for the list of headers, the content and the status code (we want to be able to repeatedly call Fetch on our instance for multiple URLs after all). Then we set up the all-important callbacks for processing the received content and header information. When we instruct libcurl to go get content for us by calling

```
curl_easy_perform(handle)
```

The library will call the callback we specify with each content block that it gets. It might very well get called multiple times (in fact the header callback will get called once for each header entry). This means that the callbacks need to handle the accumulation of the data.

The callbacks need a static function – we can’t simply call a member function of our class, and of course, if we define the functions in our class as static, they are unaware of our specific instance – hence we tell the library with CURLOPT_WRITEDATA and CURLOPT_WRITEHEADER options to

pass the ‘this’ pointer to our static functions as user data, and then we can de-reference the ‘this’ pointer to get a handle back to our instance and update the member variables. simples.

The callbacks are defined like this:

```
static size_t func(void* ptr, size_t size, size_t nmemb,
```

Which at first looks quite confusing. the reason that we have both a ‘size’ and a ‘nmemb’ (number-of-members) is that typical examples in the past used FWRITE to write the content to a file and the parameters mirror those of write making that job nice and easy. For our example, we simply multiply the number of chunks by the size of chunks and then process that amount of data. Note that the ptr to the data will not be terminated – so if you are creating or appending to a string as we are, you will need to use one of the overloaded methods that let you specify a size.

We create a vector of strings for our header entries and a simple string for the content, appending to it with each invocation.

Note that a handle to an easy libcurl instance like we are showing encapsulated in our example below is not thread safe. You can of course have multiple instances of the class (and hence libcurl handles), each on single threads – but don’t cross the streams and start sharing the handles. There is a multi interface that should allow you much more flexibility, but that is way beyond the point of this blog.

hopefully that gives you enough information to get started – we don’t claim to be experts so please feel free to point out any howlers or glaring omissions we might have made.

This entry was posted in C++ and tagged c++, cpp-netlib, http, libcurl. Bookmark the permalink.