Multiplayer Game Programming

HTTP Servers

Game 311

James Dupuis

Kahoot Review

Kahoot!

Lecture 11
Objectives

- Kahoot Review
- **HTTP Servers**
  - REST
  - Security
  - Frameworks
- Intro to lambda functions
- **HTTP Request Processing**
  - CPP REST SDK
  - HTTPListener
  - Exceptions
  - Request handling
  - Creating / Formatting Responses
  - Processing JSON

RESTful Server

- What does RESTful server mean?
  - Representational State Transfer
  - an HTTP server that processes GET, POST, PUT, & DELETE requests

- Base ideal is that all request are separate entities, and do not know or rely on previous requests states.
  - This means that if I type in the correct URI and attempt to access the API with properly formatted data I should be able to succeed regardless of the order I access the API requests.
  - With that being said, requests can still fail due to dependencies

- One dependency normally with Game Servers is that it does require a bit of security.

RESTful Security

- Typically an initial login request is silently made from a users device with a hidden ID associated with each player.
  - A first time user may have this field empty and a new ID is generated for them and returned in a response.
- That ID is used along with a timestamp of timeLoggedIn and perhaps something unique to the device such as a MAC address to generate a new session Token.
  - Games often attempt unique formats for their encryption so there is no one definitive way to assemble the Session Token.
- An encryption is typically applied to combine the data listed above into a non-human readable series of characters.
  - Example: sdf3482dsfdf234sdfsdf3442wgfs45dfv33wee3...

RESTful Security II

- The Client side application would then include this session token as a header value for each subsequent request.

```
“X-Session-Token”: “sdf3482dsfdf234sdfsdf3442wgfs45dfv33wee3...”
“UserID”: “97931893”
```

- Every request received on the server would perform a cross comparison of the data provided ensuring a few things:

Does the server have a matching entry for the session token supplied?
Does the decrypted session token match the UserID specified?
Has the timestamp of the session token expired?

- What does this solve?

Hacking – Users can’t attempt to make requests as another player
Cheating – Users can’t attempt to cheat by logging on more than one device.

RESTful Pro / Cons

- Pro:
  - Easy to debug
  - Manageable
  - Accessible
  - Easy to setup
  - Allows more connections

- Cons:
  - Slightly more overhead for data
  - Slightly less secure
  - Only the client controls communication.
  - Must be a Dedicated Server

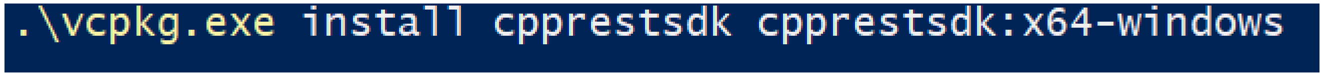
Http Server Code

- There are many HTTP server frameworks available for a variety of programming languages.
  - Each has their own pros and cons.

Language	**Frameworks**
Javascript	Node.js
Java	Spring, Spark, Restlet
C++	CPPREST, POCO, Proxygen
C#	ServiceStack, Nancy
Python	Django, web2py, pyramid

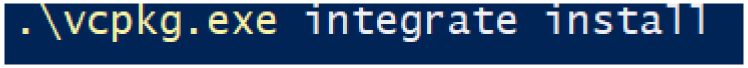
HTTP Server Setup

- For our HTTP servers we will be using CPPRest aka Casablanca
  - https://github.com/Microsoft/cpprestsdk/wiki/Getting-Started-Tutorial
- download the zip file from:
  - https://github.com/Microsoft/vcpkg
  - extract to a location on your C:\vpckg
  - enter the folder, hold shift and open a windows powershell or cmd prompt
  - Type:
    - C:\vpckg\bootstrap-vcpkg.bat
  - Once that has completed, type in the following **(this could take 30 min)**



```
.\vcpkg.exe install cpprestsdk cpprestsdk:x64-windows
```

  - Finally push the changes out to our Visual Studio Directories using:



```
.\vcpkg.exe integrate install
```

HTTP Server Setup

- To begin with, we need to include the following header:

```
#include <cpprest/http_listener.h>
```

- Next we have to ensure we are using the library for cpprest:

```
#pragma comment(lib, "cpprest_2_10")
```

- Lastly we need to specify the namespaces we will be using:

```
using namespace web;
using namespace web::http;
using namespace web::http::experimental::listener;
```

HTTP Listener

- An HTTP listener allows us to listen on a port for incoming HTTP requests.
  - This is a framework which processes the HTTP request for us and allows us to define functions to forward our message along to.
- To create a Listener we can use the following:

```
http_listener listener(L"http://72.1.214.93/SLCGame311");
```

- To listen for a specific type of request we can use the support function of an http listener:

```
void support(const http::method& method, const std::function<void(http_request)>& handler)
```

- The support function takes in a method for it’s first parameter and the second parameter is the address of a function to call to process the message.

Examples:

```
listener.support(methods::GET, handle_get);
```

- Or:

```
listener.support(methods::POST, handle_post);
```

Tasks

- Similar to our sockets, we need to ensure we open the listening server to receive incoming requests:

```
pplx::task<void> listenTask = listener.open();
```

- pplx::task is a concurrency process, this essentially handles multithreading for us.
  - Similar to a coroutine in C#
  - This is extra important because we don’t know when a request is going to come in and need to be processed.
- We can then use our listenTask to wait for incoming requests:

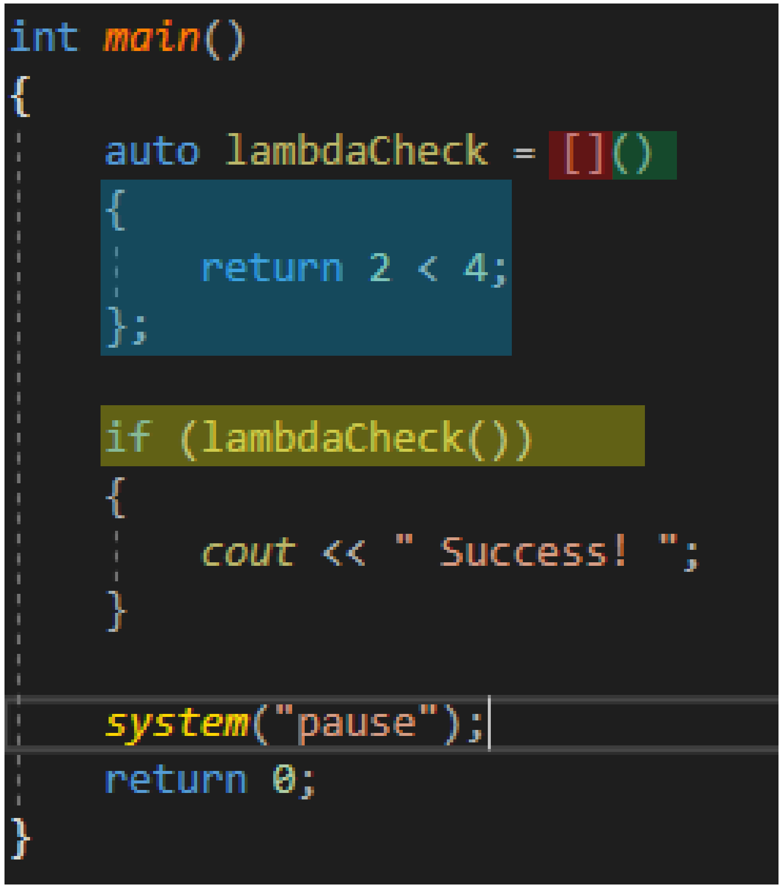
```
listenTask.wait();
```

- Concurrency tasks have another option which can precede a wait() called then().
  - then() can be used to specify a function to call when the open task completes.

```
listenTask.then(funcToCall);
```

Lambda Functions

- C++ 11 introduced a new concept known as lambda functions.
- Essentially these are embedded functions within a function.
- These are commonly used in conjunction with HTTP requests because we often need custom function implementations for processing requests and responses.
- Basic syntax has 3 separate sections:
  - **[] : Captures**
  - **() : Parameters**
  - **{} : Body**



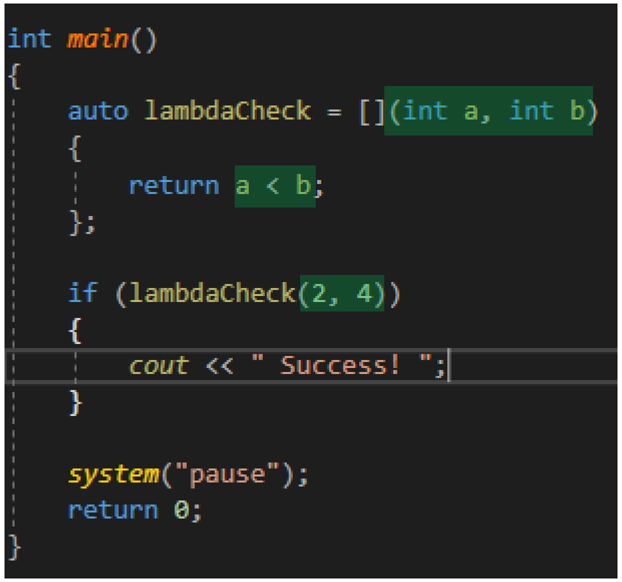
```
int main()
{
    auto lambdaCheck = []()
    {
        return 2 < 4;
    };

    if (lambdaCheck())
    {
        cout << " Success! ";
    }

    system("pause");
    return 0;
}
```

Lambda Sections

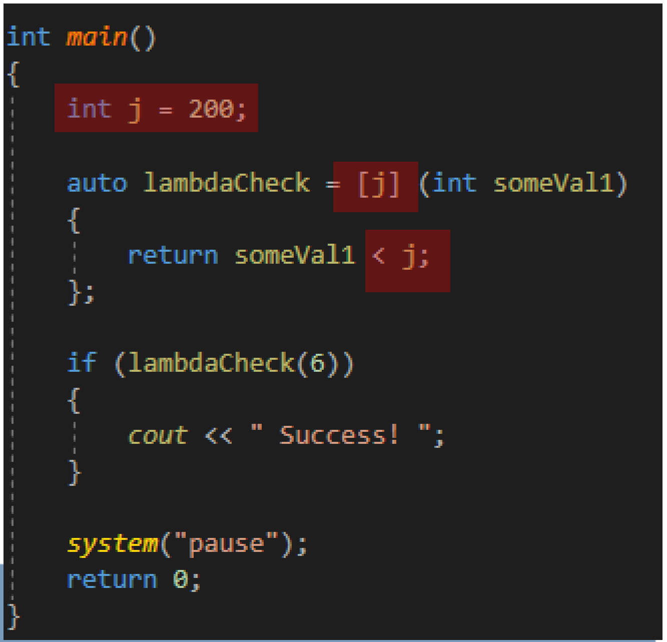
- To pass a variable to a lambda function, you need to specify the type and identifier within the parentheses (), similar to a normal function.
- The body can be used to declare a list of statements to execute when the lambda is called within code.
- The Capture [] is the new part to lamba functions, this section is similar to our parenthesis, however, we can use it to pass variables into the lambda at time of creation, instead of use.



```
int main()
{
    auto lambdaCheck = [](int a, int b)
    {
        return a < b;
    };

    if (lambdaCheck(2, 4))
    {
        cout << " Success! ";
    }

    system("pause");
    return 0;
}
```



```
int main()
{
    int j = 200;

    auto lambdaCheck = [j] (int someVal1)
    {
        return someVal1 < j;
    };

    if (lambdaCheck(6))
    {
        cout << " Success! ";
    }

    system("pause");
    return 0;
}
```

RESTful Server

- Inside our Rest SDK, we will use lambda functions with our tasks to create functions which are defined for future execution.
  - One example is when our server finished opening it’s ports for listening:

```
 listener.open()
// the then is automatically called after the open completes
    .then([&listener]()
            { wcout << "\nstarting to listen:\n");
              wcout << listener.uri().to_string().c_str();
            })
// the then function requires a function to call, it uses
// a lambda function and passes in the listener as a
   reference
```

Try Throw Catch Review

- Try/Catch statements are used to handle exceptions from a program.
- The main syntax that's important is a **try** {}
- You cannot have a try without a subsequent **catch**()
    - **Catch** cannot be left empty, it requires a single data type to catch

```
try
{
   // Code to try;
   // Throw an exception with a throw statement
   throw 0;
   // More code to try;
}
catch (int exeception)
{
   // Code to process the exception;
}
```

- Without a throw, the try catch is essentially useless, but can still exist…
- The **throw** is what sends the program into the catch segment.
    - **Throw** can be used even without a try catch, but it will cause a runtime error if not caught

Putting it all together

```
// create our http server object
http_listener listener(L"http://localhost/restdemo");
// call the get function when a GET request comes in.[More on next slide]
listener.support(methods::GET, handle_get);
try
{
     // Listener.open() can potentially throw an error, so we need to
     // handle it here. The throw will actually climb up the
     // callstack and be caught by this try.
    listener.open()
        // the then is automatically called after the open completes
        // the then function requires a function to call, it uses
        // a lambda function and passes in the listener
       .then([&listener]()
                  { wcout << "\nstarting to listen:\n");
                    wcout << listener.uri().to_string().c_str();
                  }) .wait();
     // perform a loop that will never end, all the while our
     // listeners perform requests
    while (true);
}
catch (exception const& e)
{
    // print out why we failed to start the http server ( wide chars)
    wcout << e.what() << endl;
}
```

http_request

- Before we open our connection we use the following to create a callback function to use when a GET request is made:

```
listener.support(methods::GET, handle_get);
```

- The handle_get parameter is a function pointer.
- This function is of type const *std*::*function*<void(*http_request*)>
  - This means that it requires a http_request object as a parameter defined.
- We define our handle_get function like so:

```
void handle_get( http_request request)
```

- The request object passed in gives our server all the information we need to retrieve from the request data the user submitted:
  - URI
  - Headers
  - Body (for post)
- The request itself is also used for responding.

Using http_request

- URI:

```
request.absolute_uri().to_string();
```

- Headers:
  - Detection:

```
request.headers().has(L”headername”);
```

  - Retrieval:

```
web::http::http_headers reqHeaders = request.headers();
reqHeaders[L"headername"]
```

- Body

```
request.body()
```

  - This returns a stream pointer which can be used to read the contents as binary data.
- Reponse:

```
request.reply(status_codes::OK, “Success”);
```

C++ JSON Response

- In order to process JSON objects in our C++ Server, we will need to include a header to help:

```
#include <cpprest/json.h>
```

- With that inclusion, we are able to access json::value a class used to house a JSON object.
- We can create a new json object with the following:

```
json::value JSONObj = json::value::object();
```

- We can then access/ set individual items of an object through string access
  - Remember JSON is all “KEY”: Value pairs.

```
JSONObj[L"GameOver"] = true;
```

- The request response can actually take in a JSON object directly:

```
request.reply(status_codes::OK, JSONObj);
```

HTTP Response

- We can send responses directly passing parameters for the status code and contents to the .reply function of the request as shown in the previous slide.
  - This function has many overloaded options.
  - One option is to pass in a response object.

- When working on games it’s often more ideal to create our response as an object as many of our responses will be formulated the same way containing similar header structure.
  - Remember a response also contains headers, not just the requests.
    - And a client can reject a response if the headers aren’t formed correctly.
  - More often the headers of a response are used to tell the client what type of data they are receiving and how large the data is.
- To Create a response object we can use the following:

```
http_response myResponse;
```

HTTP Response

- The http_response object contains all the fields we will need to send data properly back to our clients.
- To set a header of the response we can use the following:

```
response.headers().add(L”HeaderKey", L“headerVal");
```

- Often times we want to set the content type of the response through the headers.

```
response.headers().set_content_type("application/json");
```

- To set the body contents of a response you can use the following:

```
response.AddBody(myJsonObj);
```

  – Be careful with this function as it can change the content-type header of your response.
- The final, yet most important part to a response is the status code which you can set like so:

```
response.set_status_code(status_codes::OK);
```

- There are many response codes available inside the status_codes class

C++ JSON Request

- http_request object has an **extract_json()** function which can be used to retrieve data from a request.
  - The extract json function returns a task which contains a .**get()** function used to access the data.

```
// Create a JSON object
json::value temp = json::value::object();
request.extract_json()
    //extracts the request content into a json object
    .then([&temp](pplx::task<json::value> task)
    {
        temp = task.get();
    })
    .wait();
   // do whatever you want with 'temp' here
   // temp contain all the json stuff
```

Summary
Learned

- **HTTP Servers**
  - REST
  - Security
  - Frameworks
- Intro to lambda functions
- **HTTP Request Processing**
  - CPP REST SDK
  - HTTPListener
  - Exceptions
  - Request handling
  - Creating / Formatting Responses
  - Processing JSON