- A lambda function is a function that you can write inline in your source code.
    - Usually to pass into another function
    - Similar to the idea of a functor or a function pointer.

Lambda Functions

Lambda functions are new features in C++11 that allow a programmer to not only program in different ways but also save time when implementing particular functions. In summary lambda functions allow you too:

- create quick functions that only need to be declared locally.
- pass functions as parameters similarly to function pointers and functors.
- use variables declared within the scope function is declared in.
- Lambda functions work well with STLs (such as algorithms).

Basic Lambda syntax

Lambda functions have a kinda funny syntax that can look like an array declaration with parameters.

```
lambda_type lambda_name =
[capture_specification] (parameters) ->
return_type {body}
```

There are a few important things to remember about the declaration of lambda.

- lambda functions can be nameless. This allows lambda functions to be used once in a specific way.
- The capture specification can be empty ***(more on this later)***
- There does not have to be a parameters field or a return type
    - You can only exclude a return type if the compiler is unable to deduce the return type.
    - The default return type of lambda functions is void.

For example here is the well know HelloWorld function written as a lambda function.

```
#include
using namespace std;

int main(){
    auto func = [] {cout << "Hello, World!";};

    // call func();
    func();
```

```
    return 0;
}
```

Variable Capture.

Variable capture is probally the best things about using lambda functions. Lambda functions have the ability to reference, use and even change variables and declared outside of the scope of a lambda function. This is all done within the capture specification field `[capture_specification]`.

```
#inlcude <iostream>
using namespace std;

int main(){
    string input;

    cout << "enter a number: ";
    cin >> input;

    auto lambda_isNeg = [input] {
    return input < 0 ? "negative" : "positive
    }

    cout << "you entered a " << lambda_isNeg <<
"number" << endl;
```

```
    return 0;
}
```

The above example show how you can capture a variable using a lambda function. In the example only `input` is captured by value; however, you do not need to give a list of variables you want to capture. Instead you can capture in the following ways:

- `[=]` capture all variables declared before the lambda function. (check this out for an example)
- `[&]` similar to `=` only instead you capture all values by reference.
- `[variable_name]` like the above example, individual variables may be captured using their name.
- `[]` is the default capture method that will capture all variables as const.
    - You can also mix and match the above three ways using commas to separate different capture types.

```
...
int x = 5;
int double pi = 3.14;
char z = 'z';

[x];        /* capture only x */

[&x];       /* capture only x by reference */
```

```
[=, &pi]    /* capture all variables by
value, but pi
               by reference */

[&, z]      /* capture all variables by
reference, and
               z by value. */

[]          /* capture all variables as
consts. CHECK! */

[this]      /* you can even capture the this
pointer
            if you have not already used [=]
or [&] */
...
```

does `[=]` capture by making a copy?

But how do lambda captures work?

Well as it terns out lambdas variable capture works in the same way as a functor. The way that lambdas are implemented is by creating a small class the overloads the operator `()` , this allows lambda to act like a function. The lambda function then becomes an instance of this class and when it is constructed it takes in all surrounding

variables within its environment that are passed into the capture specification field and stores them as member variables. ***(this may need to me moved or reworded!)***

Tread carefully, the pros and cons to `[&]`

While all of this seems great you should know when it is okay to capture by different types. For example if you have a lambda function as a function parameter. If a lambda function captures values by reference, the reference will not be valid after the function returns. However, if you capture by reference you have to ability to modify and change the data.

Using lambda with the STL

To use STLs like algorithms was a tedious task before lambda functions. Lets look at `for_each` in the algorithms STL as an example:

```
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);

// before in C++03 you would have to use a for
expression to print out the values of v.
```

```
for (auto itr = v.begin(), end = v.end(); itr !=
end; ++itr){
    cout << *itr << " ";
}

// now however you can easily use the for_each.
for_each(v.begin(), v.end(), [] (int val) {cout
<< val});
```

Does this implementation not seem easier? To me it does. And as it would tern out, the for_each implementation performs at the same speed or even faster than the for expression because to can easily utilize loop unrolling.

Putting it all together, Using lambda functions!

In the examples given so far we have given the lambda function a name. This is possible because the lambda expression is typed, which also allows us to assign a lambda to a function object.

```
int main(){
    // Assign a lambda expression to a function
object.
    function<int (int, int)> func = [] (int x,
int y) {return x + y})
```

```
}
```

USING LAMBDA FUNCTIONS:

Lambda functions are mostly used as parameters for other functions, such as the for_each example above.

NESTING LAMBDAS

Yes, you can nest lambdas inside of each other! (from: www.msdn.microsoft.com)

```
#include <iostream>
using namespace std;

int main(){
    int m = [](int x)
        {return [](int y) {return y*2; }(x) + 3;
} (5);
      //|-body of first-|-----body of second-----
|
}
```

In the above example the first lambda `int m = [](int x)` is declared directly after the `int main()`. The first lambda returns the second lambda + 3. The second lambda `[] (int y)` that returns `y*2`. To understand how this works lets first break down what `(x)` and `(5)` meant. You will notice that `(5)` is outside of the scope of both lambdas, this means that the parameter of the first lambda `(int x)` will have a value of 5. `(x)` is the same only it refers to the `y = x`, and `x = 5` so `y = x = 5 -->`

y = 5.

LAMBDA IN A HIGHER-ORDER

Lambda functions that take another lambda function as a parameters and returns a lambda function. ***this needs to be explained*** from here.

```
#include <iostream>
#include <functional>

int main()
{
   using namespace std;

   // The following code declares a lambda
expression that returns
   // another lambda expression that adds two
numbers.
   // The returned lambda expression captures
parameter x by value.
   auto g = [](int x) -> function<int (int)>
      { return [=](int y) { return x + y; }; };

   // The following code declares a lambda
expression that takes another
   // lambda expression as its argument.
   // The lambda expression applies the argument
z to the function f
   // and adds 1.
   auto h = [](const function<int (int)>& f, int
z)
```

```
      { return f(z) + 1; };

   // Call the lambda expression that is bound to
h.
   auto a = h(g(7), 8);

   // Print the result.
   cout << a << endl;
}
```

NOT DONE, STILL A WORK IN PROGRESS!

other notes:

Notes for declaring lambda expressions:

- you can declare a lambda anywhere that your can initialize a variable

initialize a variable.

-

Notes for capturing variables:

- The Visual C++ compiler binds a lambda expression to its captured variables when the expression is declared instead of when the expression is called.
- **Reassignment of Captured Variables**:
  - if a variable is captured by `[=]`, then any reassignment of that variable will not effect lambda, meaning that it will use the value before the reassignment of the variable will be used by lambda because lambda makes a copy of the variable at the time of its declaration.
  - for `[&]`, you are passing by reference and because of that if you reassign the variable the newest value will be used in the lambda expression.

Notes for using lambda

- `[] (int x, int y) {}returns x + y}(5, 4)` will have x = 5 and y = 4 (possibly only when compile with /EHsc). loop at this.

source