



COP3503

Lambda Expressions

| Lambda Expressions (Lambdas)

- + Anonymous functions—functions without a name.
 - They aren't formally declared like other functions.
- + Essentially dynamic function objects—no class needed.
- + Good if you need a custom function, but only in one place
 - Keeps functionality “localized” to only where you need it

Store it, in an **auto** variable
(or use a template)

```
auto lambda = []() { cout << "Hello World!"; };  
lambda();
```

Define the lambda
expression

Call it, just like a function

| Lambda: Syntactic Sugar for Functors

- + **Syntactic Sugar**: A nicer, easier alternative for an existing language feature (or from some other language)
- + Code that isn't strictly necessary but is nice to have.
 - The code equivalent of candy—tasty, but we don't **need** it!
- + Just affects how you, the programmer, write a thing
- + Some languages (especially newer ones) are “sweeter” than others.
 - C++ has gotten sweeter over time, but was never designed to be a sugary language.

Syntactic Sugar Example: The `auto` keyword

You can write lots of code without this, but sometimes it's nice to use!
It can make some parts of the language less “sour” or “bitter”

| Lambda Expressions Syntax

```
[capture clause](parameters)->returnType(optional)
{
    // Lambda code. What does this process do?
    // I.e. write your "function" here
};
```

- + **parameters**: Just like a function, write your parameters here
- + **returnType**: Just like a function (you can omit it, and the compiler deduces the type—like using the **auto** keyword)
- + **[capture clause]**: How does the expression access variables from the surrounding scope? (more on this later – it can be empty)

Lambda Example: Sorting

```
void SortStuff(vector<int>& numbers, bool(*compare)(int, int));  
bool Ascending(int a, int b)  
{  
    return a > b;  
}  
SortStuff(someVector, Ascending);
```

What if you don't want to write this?

You need it, but it's just
"laying around" afterward.

```
// Call the function and pass it a lambda expression  
SortStuff(someVector, [](int a, int b)->bool {return a > b;} );
```

The actual lambda expression

```
[](int a, int b)->bool {return a > b; }  
// -OR-  
[](int a, int b) {return a > b; } // Return type omitted (and inferred)
```

// Spacing the lambda out can make it more readable

```
SortStuff(someVector,  
[](int a, int b)->bool  
{  
    return a > b;  
});
```

The lambda expression replaced a function that would never be used (aside from passing them as data).

This can declutter your code, remove "useless" functions or functors.

| Another Example Using `std::find_if`

- + Scenario: You want to search through some STL container for a specific value

Let's assume:
`vector<Person> persons;`
Search for: "Adam"

`std::find_if()` has you covered!

`std::find_if(startIterator, endIterator, predicate)`

`persons.begin()`

`persons.end()`

Some lambda expression...


```
// Parameters one and two might be more or less "standard"
auto result = std::find_if(persons.begin(), persons.end(),
[] (const Person& p) -> bool
{
    return p.GetName() == "Adam";
});
```

This is a "quick and dirty" function defined right here, and only accessible right here.

There's no "FindAdam()" function anywhere in your code to confuse someone else.

| Creating and Storing Lambda Variables

- + Lambda variables can be created for reuse, but they must be stored using:
 - 1 The **auto** keyword
 - 2 Templates, or `std::function` (which uses templates)
- + Why? They have a type, but... it's not meant for humans

Watch 1		
Name	Value	Type
 lambda	{...}	main::__l2::<lambda_4086357f8e2eb6296d1b1830d8d6434a>

You probably don't want to write this out...

The compiler determines all of this. You don't have to worry about it.

```
// Store a lambda expression in a variable, and again in a std::function
auto lambda = [](int num) { return num > 30; };
std::function<bool(int)> func = lambda;

// Invoke the lambda expression using two different variables
int result1 = std::count_if(numbers.begin(), numbers.end(), lambda);
int result2 = std::count_if(numbers.begin(), numbers.end(), func);
```

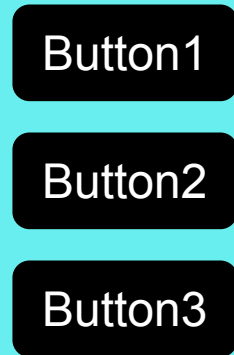
| Lambda Expression Example: Buttons

- + Imagine a UI with various Button objects.
- + Each one needs some unique “on click” functionality.

```
// An abstract base class with a virtual function
class Button
{
public:
    virtual void OnClick() = 0;
};

// Derive a new class, implement the click function
class Button1 : public Button
{
public:
    void OnClick() { /* Do some stuff */ }
};

// And then two more...
class Button2 : public Button {};
class Button3 : public Button {};
```



Button1

Button2

Button3

Four separate Button classes, **three** bits of unique functionality... that's a lot of classes!

| Buttons: Lambda Edition

```
class Button
{
    // Execute this when the button is clicked
    std::function<void(void)> _function;

    /* Plus any other necessary variables */
public:
    // Take in a lambda expression when creating a button
    Button(std::function<void(void)> func)
    {
        _function = func;
    }

    void OnClick()
    {
        _function(); // Execute previously stored lambda
    }
};
```

| Buttons: Lambda Edition

```
// Initialize a button with custom functionality
Button someButton([]() {cout << "Hello world!"; });

Button button2([]()
{
    cout << "Button 2 functionality goes here." << endl;
});

Button button3([]()
{
    for (int i = 0; i < 5; i++)
        cout << "Button 3!" << endl;
});
```

One class, 3
customized instances!

Functionality can be treated like
data, and we can set or change
it as needed.

Capturing Local Variables

- + A lambda expression can use variables declared outside of it.
- + It has to “import” those variables through the capture clause []
 - [] No access to locals (the default)
 - [=] Capture by value. Access, but no modification
 - [&] Capture by reference. Allows access/modification

These can be set
“globally” or on a
per-variable basis.

```
// Local variables outside of the lambda expression
int x = 5, y = 10;
```

```
// default, no access
[]() {
    cout << x; // Not okay
    y += 2;    // Not okay
};
```

```
// By value only
[=]() {
    cout << x;    // Okay
    y += 2;       // Not okay
};
```

```
// By reference, full access
[&]() {
    cout << x;    // Okay
    y += 2;       // Okay
};
```

```
// By value, y by reference specifically
[=, &y]() {
    x -= 5;       // Not okay
    y += 2;       // Okay
};
```

Capture Clause Example

// An example from earlier

```
auto result = std::find_if(persons.begin(), persons.end(),  
[](const Person& p)->bool  
{  
    return p.GetName() == "Adam";  
});
```

What if you want
"Adam" to be
something else?

How do we change this?
Or send something to
the lambda?

```
string userInput;  
getline(cin, userInput);
```

The capture clause "brings in" the userInput variable.

```
auto result = std::find_if(persons.begin(), persons.end(),  
[=](const Person& p)  
{  
    return p.name == userInput;  
});
```

We COULD just get
user input inside the
lambda expression...

But this is being executed
numerous times, and we don't want
user input numerous times.

| Function Pointers vs. Functors vs. Lambdas

- + Function pointers are the original – still useful, but might feel a little outdated.
- + Functors are a better alternative.
 - Syntax is cleaner, less headache.
 - You may have lots of small, trivial classes floating around.
- + Lambdas are good when you need “disposable” functions that won’t live on.
- + Fundamentally, they are all very similar.

They can make for a very different
(and powerful!) way of programming.

They can be quite confusing at first.
(You aren’t alone if you think so!)

Programming Is Always About Doing Something with DATA

- + Our programs operate on data (whether primitive types or objects).
- + Treating a set of instructions (i.e., functions) as data allows for more complex applications.
- + In the same way you might read data, you might write data.
- + ...you can pass sets of instructions as data to other programs.

Does it matter that your code READS as Foo(), specifically?

Or is it more important that the Foo() function gets called when your program needs it?

// Instead of this:

```
Foo();  
Bar();
```

// You could have this

```
FirstFunction(); // which COULD be Foo()  
SecondFunction(); // which COULD be Bar()
```

// Or this! Call each function, however many there are...

```
for (int i = 0; i < numberOfFunctions; i++)  
    functions[i]();
```

These two lines can be reused over and over, for any number of functions.

| Recap

- + Lambda expressions are anonymous, temporary functions we can create on an as-needed basis.
- + They're used in the same situations as function pointers and functors.
 - But don't have to be created in advance
 - No “lingering” functionality after they've been used
- + Lambdas must be stored in auto variables or templates.
 - The compiler determines their types.
- + Often easier to use than function pointers and functors.
 - Syntactic sugar for creating custom operations—a “sweeter” way of achieving the same results.
- + Lambdas can allow for a lot of dynamic behavior in programs.
 - They're very powerful, but can be challenging to work with (so can function pointers and functors).



| Conclusion



Placeholder for the instructor's welcome message. Video team, please insert the instructor's video here.



Thank you for watching.