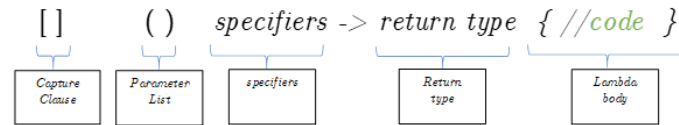


Chap 16 - Lambda Expressions (C++17 version)

Lambda expressions were created in C++11 and have become a major part of the language. Knowing how Lambdas work and what is going on under the hood is important. Note we are going to learn Lambdas from both C++14 and C++17 so make sure your compiler/linker options are set correctly.



The syntax of a Lambda expression is made up of 5 parts:

1. **Capture Clause** (aka the Lambda introducer) – the capture clause is used to both specify a lambda function and it contains a list of variable from outside the Lambda that the Lambda may use. Variables may be passed by value or by reference.
2. **Parameter list** (*optional*) – these are the arguments that are passed to the lambda. They differ from the variables passed from the Capture Clause. Capture Clause variables are used in the creation of the Lambda; the parameters in the parameter list is used in the invocation of the Lambda.
3. **Specifiers** (*optional*) – these are specifiers that modify the behavior of Lambdas
4. **Return Type** (*optional*) – Normally the compiler will deduce the return type but you can specify it here.
5. **Lambda Body** - The actual body of the code

The following exercises will cover all of these parts.

1. Capture Clause - In your main function create three int variables x, y and z. Also make a global variable int named 'glob'
 - a. Create a lambda that copies x and y in the capture clause but NOT z. Return the sum of x and y.
 - b. Create a lambda that copies x by reference and y by value (again not Z). Set x equal to the sum of x and y.
 - c. Create the following Lambda expression

```
x = 1; y = 2;
auto exref = [&x, y]() {return x + y; };
cout << exref() << '\n';
x = 10; y = 20;
cout << exref() << '\n';
```

The return from exref is printed twice... what do you expect the values to be? Be prepared to explain why it printed them. Try it in <https://cppinsights.io/> if you need help.

- d. Take problem b. and run it in cppinsights.io. How are x and y declared? Be prepared to explain this in class.

- e. Here is a question... can you capture a global variable? Create a lambda that attempt to capture your global variable named 'glob' (that is put it in the capture clause like so:

```
[glob]() {return glob+1;}
```

Why do you think this doesn't work? Notice the compilation error. Now remove *glob* from the capture clause but still return *glob*+1. Does it work?

- f. `std::cout` is a variable... why don't I need to capture it when I use it in a Lambda?
- g. Inside of your main function create a static int variable called 'stat'. Try and capture the variable *stat* like so:

```
[stat]() {return stat + 1;}
```

What happens? Do you get an error? What happens if you remove *stat* from the capture clause?

- h. Try the following code:

```
int x = 3;
auto add = [x](int y) {x = x + 1; return x + y;};
```

Does it compile? By default lambdas do not allow you to modify the capture variables even if they are by value. We override this default behavior in exercise 2.

- i. A feature introduced in C++14 was the ability to create a new variable and initialize it in the capture clause. For example:

```
x = 3; y = 9;
auto ex1g = [w = 5 + x * y]() {return w/2; }; // w=32 returns 16
cout << ex1g() << '\n';
```

The compiler will type deduce that *w* is an integer and *w* to $5 + x * y$ ($5 + 3 * 9 = 32$). The variable *w* can be referenced in the lambda body but... not *x* and *y* in this case. Put this code onto cppinsights.com and see the resulting code.

So try the following initializations in the capture clause:

- i. A variable initialized to 42
- ii. A variable initialized to "hello" that would be interpreted as a `const char *`
- iii. A variable initialized to "hello" that would be interpreted as a string object

- j. A capture clause that is empty defines a special type of lambda expression called a *stateless lambda*. There are a couple of nice things about stateless lambdas... but one of them is that the compiler can convert them to function pointers. This allows you to do things like create a array/vector/deque of lambdas easily. If there is a capture clause this can still be accomplished (as we will see later) but there is a larger memory cost overhead.

Here are some examples:

```
// a function pointer that returns an int
int (*fp)();
fp = []() {return 42; };
cout << fp()<<'\n';

// a function pointer that takes two ints as arguments and returns an int
int (*fp2)(int a, int b);
fp2 = [](int a, int b) {return a+b; };
cout << fp2(4,5) << '\n';

// Same as above but creates a new type 'fptype' first for readability
using fptype = int(*)(int,int);
fptype fp3 = [](int a, int b) {return a+b; };
cout << fp3(4,5) << '\n';
```

Notice the 3rd example creates a new type called *fptype* that makes the code a little more readable. This would also allow us to easily create a container of lambdas and execute each one as follows

```
vector<fptype> funcptrs = {
    [](int i,int j) {return i + 2*j; },
    [](int i,int j) {return 3*i + 2*j; },
    [](int i,int j) {return 7*i + 9*j; },
};

for (auto myfp:funcptrs){
    cout << myfp(2,3)<<'\n';
}
```

So try the following

- i. Create a type called *fptype* that is a function pointer that returns an int and takes one int as an argument
- ii. Create a vector of *fptype* and add 3 lambda expressions that take an int as an arg and returns an int.
 - The first lambda returns 2 * the argument
 - The second lambda returns the argument squared
 - The third lambda returns the argument divided by 4
- iii. Make an integer variable called *test* and assign it to 10. Apply each lambda function in the vector to *test* and assign *test* as the return value after each call (so after the first lambda *test* will equal 20).
- iv. Print out the final value of *test*

- k. Capturing the *this* pointer – In the past using a Lambda in a class method was a little tricky. It has become much simpler but there are still some gotchas.

First it is important to note that there are two ways to capture the *this* pointer – you can capture the pointer directly like so: `[this]() { ... }` OR copy a dereference of the pointer `[*this]() { ... }`.

There is a BIG difference between the two:

- `[this]() { ... }` just captures the pointer to the current object
- `[*this]() { ... }` captures a COPY of the object. That is a brand new copy of the object is made.

Let's look at an example. Type in the following class:

```
class A {
public:
    int x;
    A(int x = 4) : x(x) { cout << "construct A\n"; }
    A(const A& a) : x(a.x) { cout << "a new copy was made\n"; }
    ~A() { cout << "A deleted\n"; }
    auto test_capture_this() {
        auto testme = [this]() { return x; };
        return testme;
    }
    auto test_capture_this_copy() {
        auto testme = [*this]() { return x; };
        return testme;
    }
};
```

Do the following in your main code:

- Create an object of type A (call it 'atest') – it should print out 'construct A'
- Set the variable 't1' to equal `atest.test_capture_this()`. Does it print out anything... in other words is a new object created or copied?
- Set the variable 't2' to equal `atest.test_capture_this()`. Does it print out anything... in other words is a new object created or copied? How many times? Why do you think it did this?
- Making copies of what the object that *this* points too is a good feature... it can be very useful in multithreaded applications... but care must be taken especially if the object is very expensive to copy. Also take note what happens when you add in the following code:

```
auto t3 = t2;
auto t4 = t2;
auto t5 = t2;
auto t6 = t2;
auto t7 = t2;
```

- v. So in general you will *normally* want to capture the *this* pointer without dereferencing as you won't normally want a copy. However, capturing the *this* pointer without copying can also be VERY dangerous as well. Try the following code:

```
auto badthing() {  
    A atest;  
    return atest.test_capture_this();  
}  
  
int main() {  
    auto bt = badthing();  
    cout << bt() << '\n';  
}
```

1. What is the result?
2. Explain WHY this is happening.
3. Would this happen if you had returned `atest.test_capture_this_copy()` ? Try it as well and see.

That covers it for the capture clause. I recommend if you have any issues to ask or try it in cppinsights.io. It will explain a great deal about what is happening 'under the hood'.

2. Specifiers –

- a. Put the following code in your main function:

```
int x = 3;
auto add = [x](int y) {x = x + 1; return x + y;};
```

Does it compile? Nope? Why not? If we comment out the offensive code ($x=x+1$) and run this in cppinsights.io we can determine what the problem is. (hint: look at how the operator() method is declared).

- b. A way to solve this problem is to add the *mutable* specifier like so:

```
auto add = [x](int y) mutable {x = x + 1; return x + y;};
```

How does this change the code in cppinsights.io? So looking at this what does the *mutable* specifier do?

- c. For problem a. try changing the value of y. Is mutable necessary? Why or why not?
d. Another specifier is *noexcept*. This is an optimization specifier and should be used when your code does NOT throw an exception. This will allow the compiler to not check for exceptions on the return. It will make the code faster especially if you call it often.

From this point forward make all your lambdas ***noexcept*** unless they DO throw an exception (say you are working with strings...).

```
auto lamnoexcept = [](int i, int j) noexcept {return i+j;};
```

3. Parameter lists – these are ‘*technically*’ optional if you are not passing an argument. However, if you are using a specifier such as *mutable* or *noexcept* then it needs to be there. Therefore... you basically need it all the time even if it is empty 😊.

Lambda Parameter lists are very similar to function argument lists. For example taking default arguments:

```
auto defarg = [](int i = 5) {return i * 2;};
cout << defarg() << '\n';
cout << defarg(42) << '\n';
```

In C++14 some big changes were made to Lambdas parameter lists. The biggest are *Generic Lambda*. Generic Lambdas can perform type deduction on the parameter list at compile time... For example by specifying the arguments as ‘*auto*’ the compiler deduces the type at run time.

```
auto gen_add = [](auto a, auto b) {return a + b;};
cout << gen_add(4, 5) << '\n'; // print 9
string a = "hello", b = "world";
cout << gen_add(a,b) << '\n'; // print helloworld
```

Notice gen_add can take ints and add them and take strings and apply operator+(). If this reminds you of template functions that is because that is exactly what is happening. The compiler makes a number of template functions out of the operator() method (see <https://cppinsights.io/s/5fec5a43>)

Generic lambdas become more interesting when passed to other template functions that use them as arguments. For example, the lambdas below can be used to sort and print any type that is sortable and printable.

```
vector<int> vi = { 343,231,13,1332,76 };
sort(vi.begin(), vi.end(), [](const auto& a, const auto& b) { return a < b; });
for_each(vi.begin(), vi.end(), [](const auto& x) { cout << x << '\n'; });
```

One other thing to note... not all the arguments in a generic lambda need to be type deduced. You can specify some argument types and for others use *auto*.

- a. Do the following
 - i. Create a lambda that takes a string and capitalizes the first element (note: don't perform bounds checking but do make sure it throws an exception if it is empty)
 - ii. Create a vector of strings and initialize it to 4-5 various strings
 - iii. Create a generic lambda called *myapply()* that takes two generic arguments. The first argument will be an iterable container. The second will be a callable object (function/lambda/functor). This lambda will apply the callable object to each element in the container
 - iv. Pass the vector of strings and the lambda that performs capitalization to *myapply()*
 - v. Now create a vector of ints and assign them to values 0-10 and create a lambda that takes a ref to an int and multiplies it by 5. Use *myapply()* to apply the function to each element of the vector of ints
- b. Make a Generic Lambda called *myaccumulate()* that takes 3 arguments: a begin iterator, an end iterator, and initial value.

Have *myaccumulate* sum all the values from the begin to the end iterators. Pass a vector of ints as a test.

- c. Pass the vector of strings you created in problem a) above to *myaccumulate()*. It works but what does it do?
- d. Make a Generic lambda called *swap()* that takes two arguments and swaps them.
- e. In Algorithms library there is a *std::replace()* template function. Write your own *replace* using Generic Lambda.

4. Returning a Lambda – Perhaps one of the coolest features is returning a lambda expression from a function (or another lambda) such that you've built a brand new lambda.

- a. Create a **template** function called *compose()* that takes two arguments: *f1* and *f2*. The function returns a new lambda function whose body returns the value of *f1(f2(x))* where *x* can be any type. In other words this should work:

```
auto oo= [](auto i) {return i + 1; };
auto pp= [](auto i) {return i * 2; };
auto oopp = compose(oo, pp);
std::cout << oopp(5) << '\n'; // returns 11
```

- b. Create a Generic Lambda that does the same thing as your *compose* function in a).
- c. Create another lambda that takes an argument and returns *i/4*. We want to compose a new lambda that divides the return value of *oopp()* by 4. In one line of code make that happen.

- d. Make a lambda called `size()` that takes an `size_t` as an argument(call it `X`). What it returns is a new Generic Lambda that takes one argument(call it `S`). It will return true if `S.size() == X`.

For example:

```
auto size = [](size_t x) { ... };
auto size5 = size(5);
auto test = size5(string("arjun"));
```

test will be true

- e. Make a lambda called `startsWith()` that takes a character and returns a new lambda. The new lambda takes a string argument and returns true if it starts with the character, For example:

```
auto startsWith = [](char c) {.... };
auto startsWithJ = startsWith('j');
test = startsWithJ(string("joe"));
```

test will be true

- f. Make a lambda called `endsWith()` that takes a character and returns a new lambda. The new lambda takes a string argument and returns true if it ends with the character, For example:

```
auto endsWith = [](char c) {....};
auto endsWithD = endsWith('d');
test = endsWithD(string("toad"));
```

test will be true

- g. Now make a Generic lambda called `bool_and()` that takes two arguments (call them `f1` and `f2`). It returns a new Generic lambda that takes one argument(call it `X`). The new lambda function returns `f1(x) && f2(x)`. For example:

```
auto bool_and = [](auto f1, auto f2) {....};
auto d_and_6 = bool_and(startsWith('d'), size(6));
test = d_and_6(string("dennis")); // "dennis" starts with a 'd' and is 6 chars long
```

test will be true


- h. Now create a vector of strings that is initialized to “dennis”, “arjun” and “jared”. Iterate through the vector and print out the strings that are of size 5, start with a ‘j’ and end with a ‘d’. For example:

```
vector<string> vs = { "dennis", "arjun", "jared" };
for (auto &s:vs){
    if (test_5_j_d(s)) cout << s << '\n';
}
```

This will print out ‘jared’.

5. Do exercise 6 on pg 1211
6. Another crazy C++ acronym... IIFE - Immediately-Invoked Function Expression. IIFE simply means to invoke the lambda at the same time as creating it (in the same line). For example the two lines of code look almost the same but one returns a lambda and the other returns the value 42 by invoking the lambda (notice the () at the end).

```
auto a_lambda = []() {return 42; }; // a lambda
auto a_value = []() {return 42; }(); // the value 42
```



This is REALLY unreadable code... a preferred method is to call std::invoke:

```
auto a_value2 = std::invoke([]() {return 42; }); // the value 42
```

This makes it very clear that we are calling the lambda and assigning its return value.

- a. Invoke a lambda that takes no arguments.
 - b. Invoke a lambda that takes two integer arguments and returns the sums
 - c. Invoke a lambda that concatenates two strings
7. Recall that stateless lambdas (those with empty capture clauses) can be turned into function pointers. We had an example where we made a vector of them. However lambdas with elements in the capture clause **cannot** be cast to function pointers. To make that work we need to use the function wrapper from <functional>.

To illustrate this try the following code

```
auto fwrappers() {
    int ijunk = 42;

    using fptype = int (*)(int, int);
    fptype fp = [](int a, int b) {return a + b; };
    cout << fp(4, 5) << '\n';

    fptype fp2 = [ijunk](int a, int b) {return ijunk + a + b; };
}
```

You should get a compilation error on the last line. To solve this we must wrap the lambda in a function wrapper class like so:

```
using fp2type = function<int(int, int)>;
fp2type fp2 = [ijunk](int a, int b) {return ijunk + a + b; };
cout << fp2(1,2) << '\n';
```

Notice the syntax for the function wrapper... you give it the return type and then the arguments in parenthesis.

- a. Go back to problem 1.j. that takes an array of stateless lambdas and make the lambdas capture a integer variable. Make the vector accept the stateful lambdas.