Lamdba and Functional Programming

FUNCTIONAL PROGRAMMING OVERVIEW

- In traditional OOP, developers are accustomed to programming in the imperative/procedural style.
- Functional programming involves composing a problem as a set of functions to be executed.
- Functional programming avoids state and mutable data, and instead emphasizes the application of functions.
- You define the input to each function, and what each function returns.

LAMBDA EXPRESSION

- A lambda expression (lambda) is a way of defining an anonymous function object (a closure) at the location where it's invoked or passed as an argument to a function.
- Lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous functions.
- With lambdas, functions are promoted to full variable status.
  - Replacement for function objects
  - Invokable with an operator()
  - Perfect for STL algorithms
  - The syntax can range from simple to complex

LAMBDA SYNTAX

```
[ capture-list ] ( params ) -> ret { body }
[ capture-list ] ( params ) { body }
[ capture-list ]  (){ body }
```

The [] identifier, called the capture specifier, is the clearest indication of a lambda.

LAMBDA EXPRESSION EXAMPLE

- Following is a simple lambda that is passed as the third argument to the std::sort() function

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
        // Lambda expression begins
        [](float a, float b) {
            return (std::abs(a) < std::abs(b));
        } // end of lambda expression
    );
}
```

LAMBDA PARAMETER LIST

- Lambdas can both capture variables and accept input parameters.
- A parameter list (lambda declarator) is optional and in most aspects resembles the parameter list for a function.

```
auto y = [] (int first, int second) -> int
{
    return first + second;
};
```

HELLO LAMBDA

```
int main{

  // func is a variable of a compiler-inferred type
  auto func = [] () { cout << "Hello lambda!" << endl; };
  // now call the function via func
  func();
  // call anonymously
  ([] () { cout << "Hello lambda"; })();
}
```

FIRST CLASS OBJECTS

- Functional programming treats functions as first-class objects.
- As discussed, this means lambdas as a variable, return type, or even a function parameter.
- Alternatively, lambdas can be defined as a std::function type.

```
auto f1() {
    return [](->int {return 42;};
}
void f2(std::function<int()> func) {
    cout << func() << endl;
}

int main(int argc, char* argv[]) {
    cout << f1()() << endl;
    f2([]{return 42; });
    return 0;
}
```

STD::FUNCTION

The std::function type can be used as a parameter or return type for lambdas.

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


std::function<int(void)> Func(){
    static int a = 0;
    return [] {return ++a; };
}
int main() {
    auto a = Func();
    auto b = Func();
    cout << a() << " " << b();
    // What will be displayed?
};
```

VARIABLE CAPTURE

- Lambdas can capture variables outside the scope of the body of the lambda. As mentioned, [] is the capture specifier.
  - You can capture by value or reference
  - You can capture as read-only or mutable
  - You can capture specific variables or in general

VARIABLE CAPTURE SYNTAX

[] Capture nothing

[&] Capture variable by reference

[=] Capture variable by making a copy

[=, &foo] Default capture any variable by copy, but specifically capture variable foo by reference.

[this] Capture the *this* pointer of the surrounding class

MUTABLE LAMBDAS

Use “=“ to capture by value.

Variables captured by value are read-only.

The **mutable** modifier makes capture by value variables editable *inside the lambda*.

In contrast, capture by reference makes the outside variable editable.

WHAT IS PRINTED TO THE CONSOLE?

```
int main(){
    int a = 5;
    int b = 10;
    [=] () mutable{
        int temp = a;
        a = b;
        b = temp;
       std::cout << a << " " << b << std::endl;

    }();
    std::cout << a << " " << b << std::endl;
   return 0;
}
```