Lambdas up close

```
. Lambdas:
    .#1: Lambdas as callbacks

    .#2: Capturing by value under the hood

    .#3: Capturing by reference under the hood

    .#4: Mixing it up

     #5: Capturing the this pointers
```

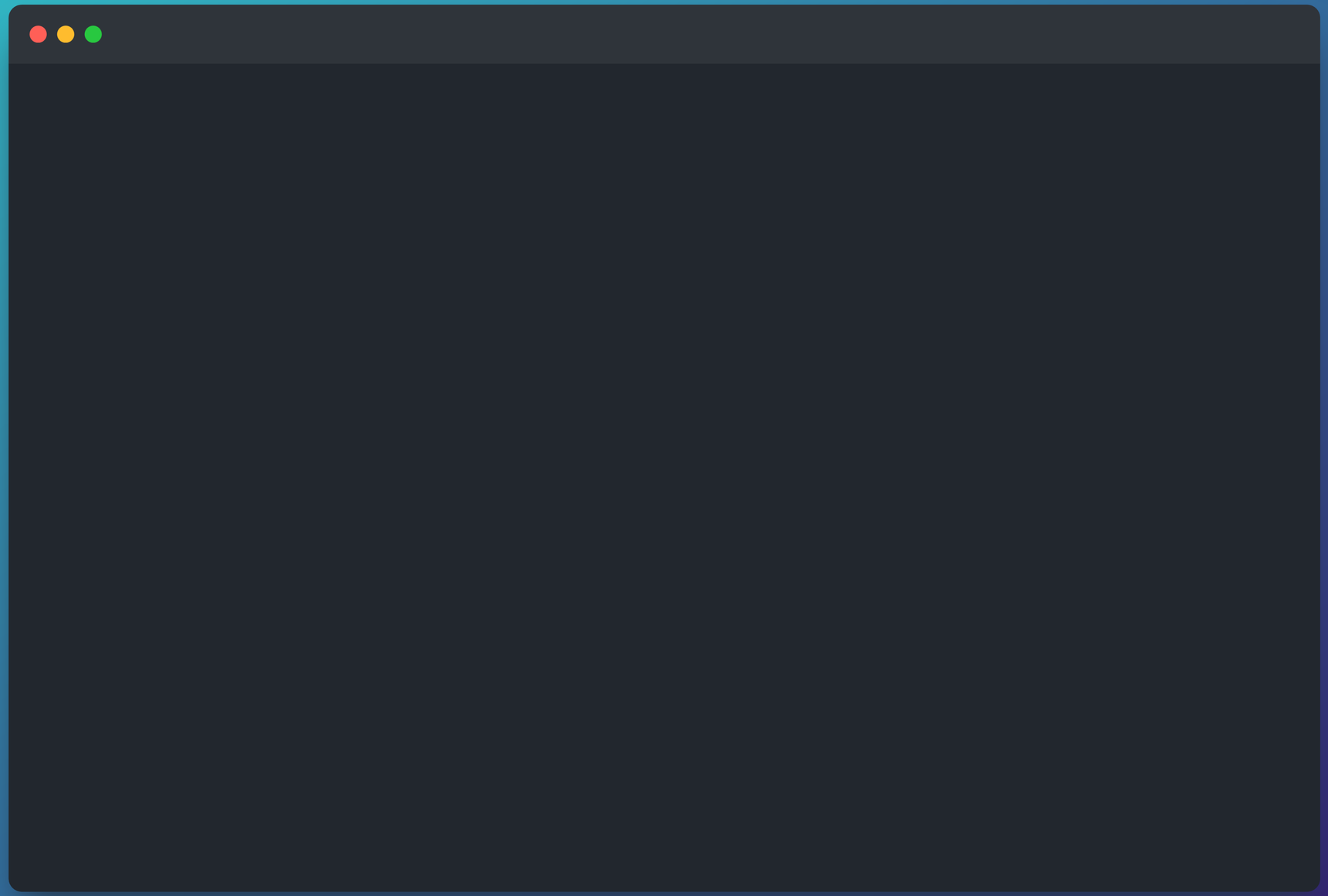
Lambdas as callbacks

```
export std::string &modify(std::string &str_param, char (*modifier)(const char &)) {
    for (size_t i{}; i < str_param.size(); ++i) {
        str_param[i] = modifier(str_param[i]); // Calling the callback
    }
    return str_param;
}

// Modifying a BoxContainer of strings
export BoxContainer<std::string> &modify(BoxContainer<std::string> &sentence, char (*modifier)(const char &)) {
    for (size_t i{}; i < sentence.size(); ++i) {
        // Code below relies on get_item() to return a reference
        // Loop through the word modifying each character
        for (size_t j{}; j < sentence.get_item(i).size(); ++j) {
            sentence.get_item(i)[j] = modifier(sentence.get_item(i)[j]);
        }
    }
    return sentence;
}

export std::string get_best(const BoxContainer<std::string> &sentence, bool (*comparator)(const std::string &str1, const std::string &str2)) {
    std::string best = sentence.get_item(0);
    for (size_t i{}; i < sentence.size(); ++i) {
        if (comparator(sentence.get_item(i), best)) {
            best = sentence.get_item(i);
        }
    }
    return best;
}
```

Lambdas as callbacks

```cpp
//Usage
std::string str{ "Hello" };

auto encrypt = [](const char &param) {// Callback function
  return static_cast<char>(param + 3);
};

auto decrypt = [](const char &param) {// Callback function
  return static_cast<char>(param - 3);
};

// Modifying through callbacks.
fmt::println("Initial : {}", str);
fmt::println("Encrypted : {}", lambdas_01::modify(str, encrypt));
fmt::println("Decrypted : {}", lambdas_01::modify(str, decrypt));

fmt::println("---------");

// Passing lambda functions directly
fmt::println("Initial : {}", str);
fmt::println("Encrypted : {}", lambdas_01::modify(str, [](const char &param) {// Callback function
  return static_cast<char>(param + 3);
}));
fmt::println("Decrypted : {}", lambdas_01::modify(str, [](const char &param) {// Callback function
  return static_cast<char>(param - 3);
}));
```

Lambdas as callbacks - Demo time!



Capturing by value under the hood

```
int a{ 7 };
int b{ 3 };
int some_var{ 28 };// Not captured by the [=] lambda, so it won't set up a member var
                   // for this
double some_other_var{ 55.5 };// Not captured by the [=] lambda, so it won't set up
                              // a member var for this.

// Capture by value
auto func = [a,b](int c, int d) {
    fmt::println("Captured values : ");
    fmt::println("a : {}", a);
    fmt::println("b : {}", b);


    fmt::println("Parameters : ");
    fmt::println("c : {}", c);
    fmt::println("d : {}", d);
};
func(10, 20);
```

Capturing by value under the hood

```cpp
// The compiler-generated functor for the lambda
class CompilerGeneratedFunctor {
private:
    int a;
    int b;

public:
    // Constructor to initialize the captured values
    CompilerGeneratedFunctor(int captured_a, int captured_b)
        : a(captured_a), b(captured_b) {}

    // The call operator represents the lambda's functionality
    void operator()(int c, int d) const {
        fmt::println("Captured values : ");
        fmt::println("a : {}", a);
        fmt::println("b : {}", b);

        fmt::println("Parameters : ");
        fmt::println("c : {}", c);
        fmt::println("d : {}", d);
    }
};
```

Capturing by reference under the hood

```
int a{ 7 };
int b{ 3 };
int some_var{ 28 };// Not captured by the lambda, so it won't set up a member var  for this
double some_other_var{ 55.5 };// Not captured by the lambda, so it won't set up   a member var for this.


// Capturing a few variables by reference
auto func = [&a, &b](int c, int d) {
    ++a;// Modifying member vars allowed by default.
    fmt::println("Captured values : ");
    fmt::println(" a : {}", a);
    fmt::println(" b : {}", b);

    fmt::println("Parameters : ");
    fmt::println(" c : {}", c);
    fmt::println(" d : {}", d);
};

func(10, 20);
```

Capturing by reference under the hood

```
// The compiler-generated functor for the lambda
class CompilerGeneratedFunctor {
private:
    int& a; // Captured by reference
    int& b; // Captured by reference

public:
    // Constructor to initialize references to the captured variables
    CompilerGeneratedFunctor(int& captured_a, int& captured_b)
        : a(captured_a), b(captured_b) {}

    // The call operator represents the lambda's functionality
    void operator()(int c, int d) const {
        ++a; // Modifying 'a' through the reference
        fmt::println("Captured values : ");
        fmt::println(" a : {}", a);
        fmt::println(" b : {}", b);

        fmt::println("Parameters : ");
        fmt::println(" c : {}", c);
        fmt::println(" d : {}", d);
    }
};
```

Mixing in up

```cpp
int a{ 10 };
int b{ 11 };
int c{ 12 };
int d{ 13 };

// Code1 : Mix by value and by ref
auto func1 = [a, &b](int x, int y) {

};

// Code2 : All by value, a by reference
auto func2 = [=, &a](int x, int y) {

};

// Code3 : All by reference, a by value
auto func3 = [&, a](int x, int y) {

};

// Code4 : capture all = and & must always  come first
/*
auto func4 = [a,b,&] (int x, int y){ // Compiler Error

};
*/
```

Mixing in up

```
// Code5 : capture all = and & must always  come first
/*
auto func5 = [a,b,=] (int x, int y){ // Compiler Error
};
*/


// Code6 : Can't prefix vars captured  by value with =
/*
auto func6 = [=a,=b] (int x, int y){ // Compiler Error
};
*/


// Code7 : If you use =, you're no longer allowed to capture any other variable
// by value, similarly, if you use & , you can't capture any other variable
//  by reference. Some compilers may give a warning, others an error.

/*
auto func7 = [=,&b,c] (int x, int y){ // Compiler Error/Warning
};

auto func8 = [&,b,&c] (int x, int y){ // Compiler Error/Warning
};
*/
```

Capturing the this pointer

```
/*
        #5: Capturing the this pointers
            . You need to capture the this pointer to access member variables and member functions.
            . You can also capture all by value and you'll get the this pointer as part of the package.
*/
```

Capturing the this pointer

```
export class Item {
public:
    Item(int a, int b) : m_var1{a}, m_var2{b} {}

    void some_member_func() {
        auto func = [this]() { fmt::println("member vars : {} , {}", m_var1, m_var2); };
        func();
    }

private:
    int m_var1;
    int m_var2;
};
```