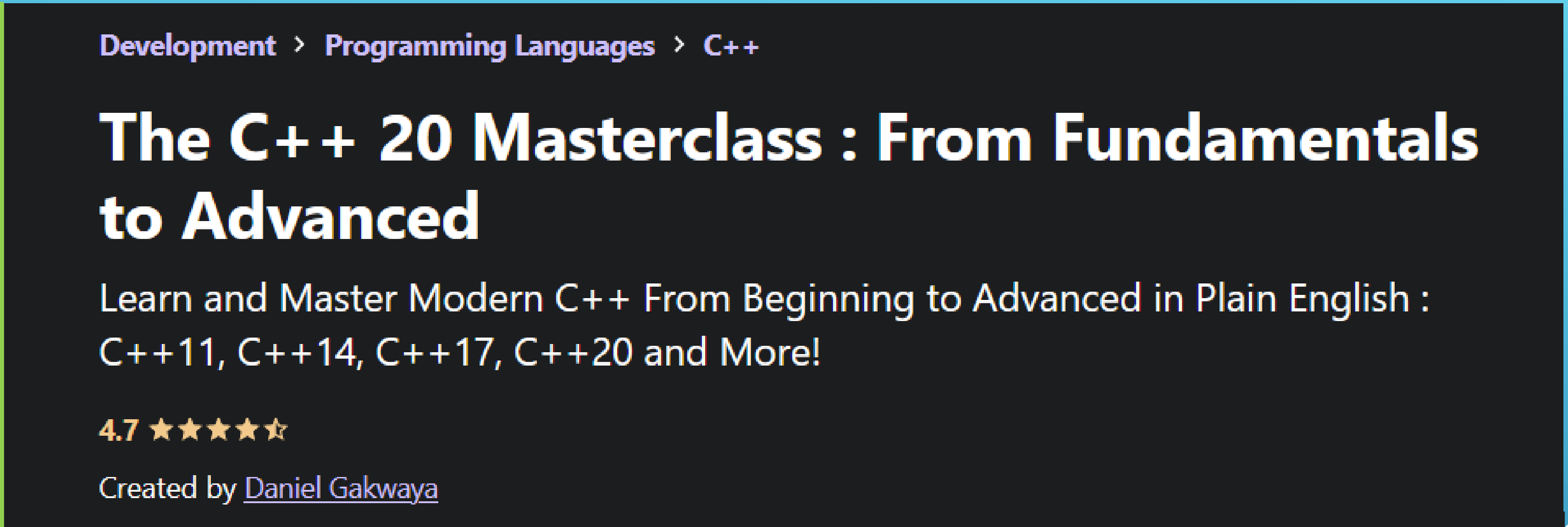
Slides



Development › Programming Languages › C++

The C++ 20 Masterclass : From Fundamentals to Advanced

Learn and Master Modern C++ From Beginning to Advanced in Plain English : C++11, C++14, C++17, C++20 and More!

4.7 ★★★★⯪

Created by Daniel Gakwaya

Section : Function Like Entities

Slide intentionally left empty

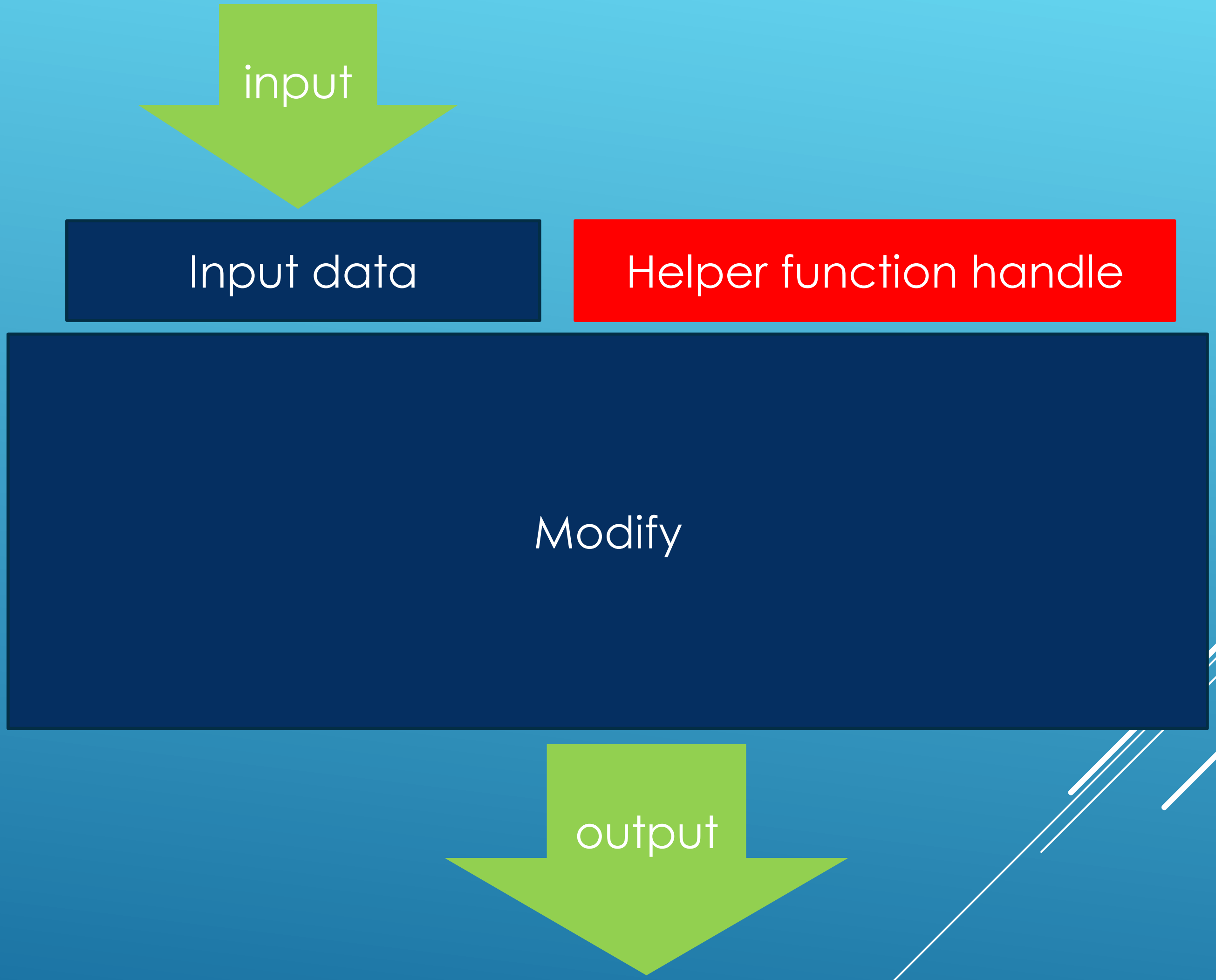
Function Like entities : Introduction

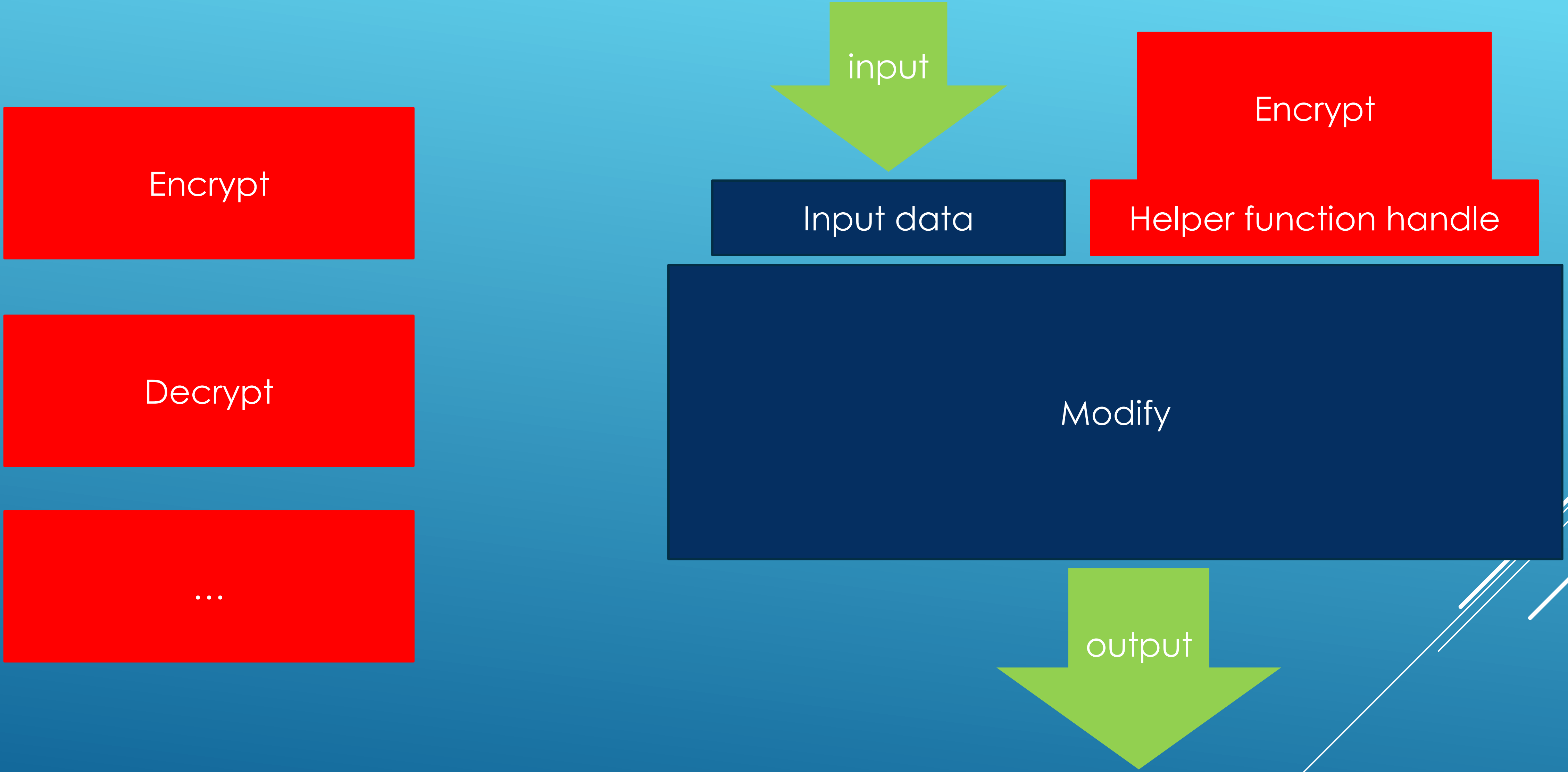
- Things that can work as functions in C++
- Taking input
- Doing something in the function body
- Returning values

Encrypt

Decrypt

...





Encrypt

Decrypt

...

input

Input data

Encrypt

Helper function handle

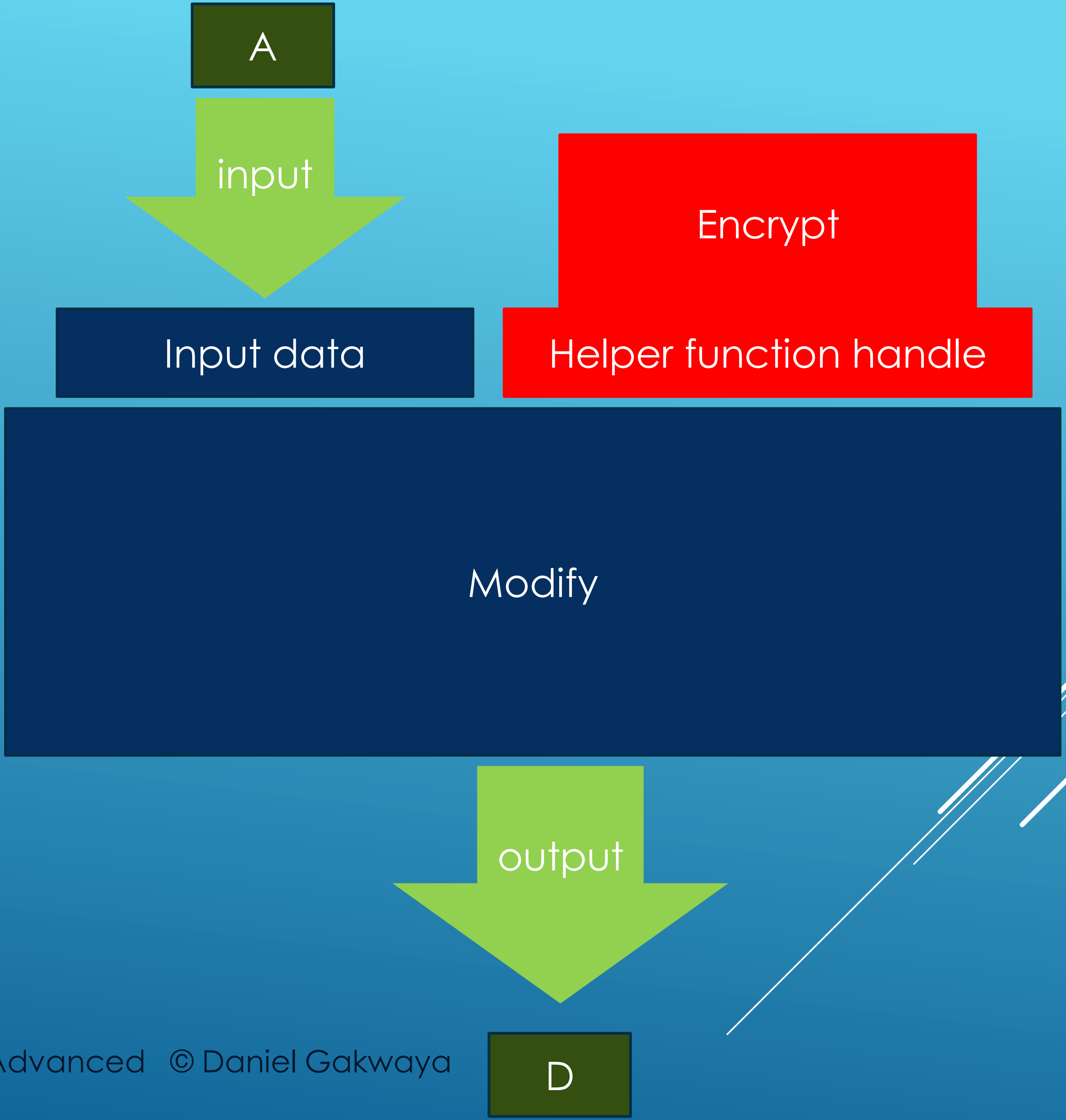
Modify

output

Encrypt

Decrypt

…



A

input

Input data

Encrypt

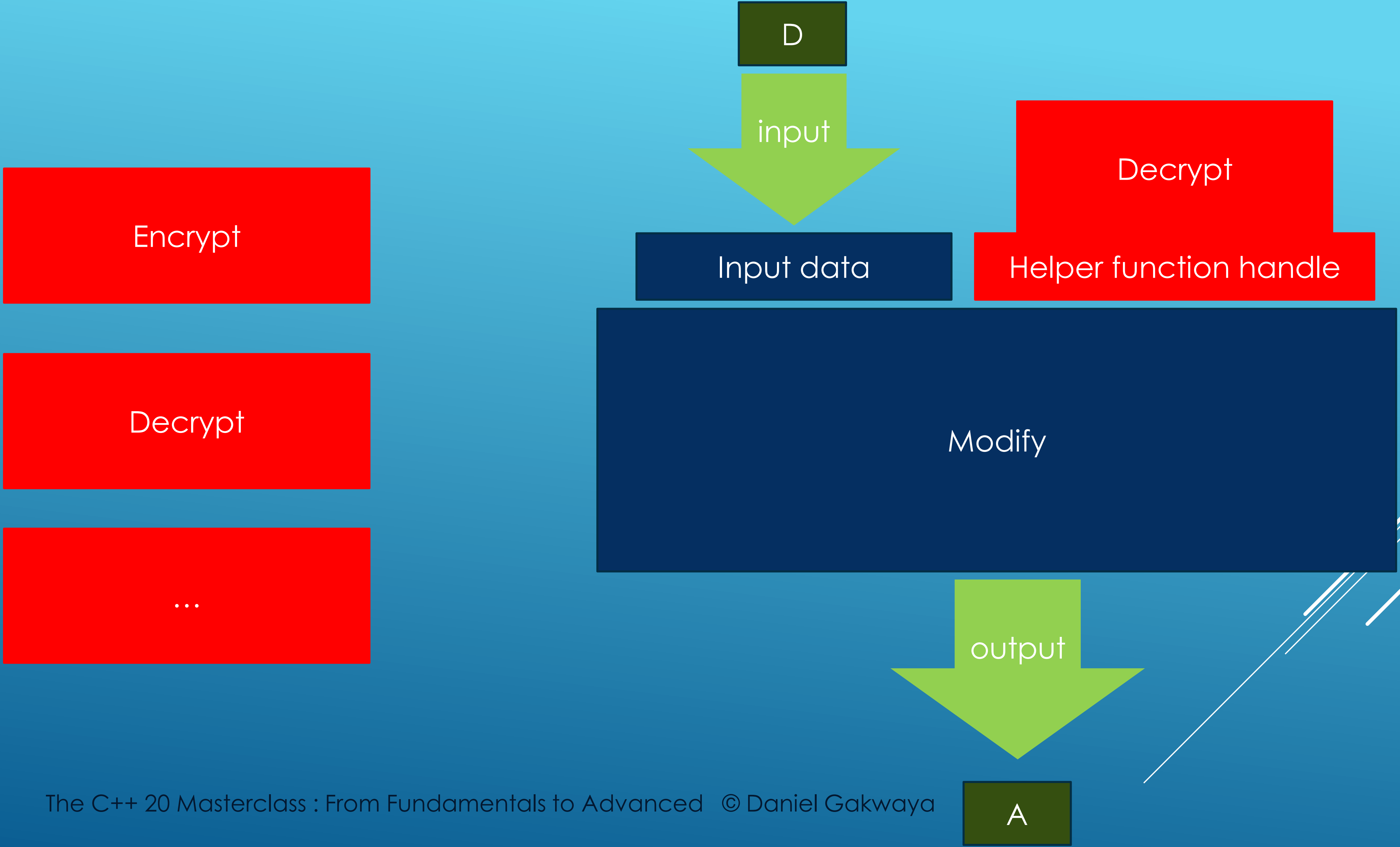
Helper function handle

Modify

output

D

ABCDEFGHIJKLMNOPQRSTUVWXYZ



Encrypt

Decrypt

...

D

input

Input data

Decrypt

Helper function handle

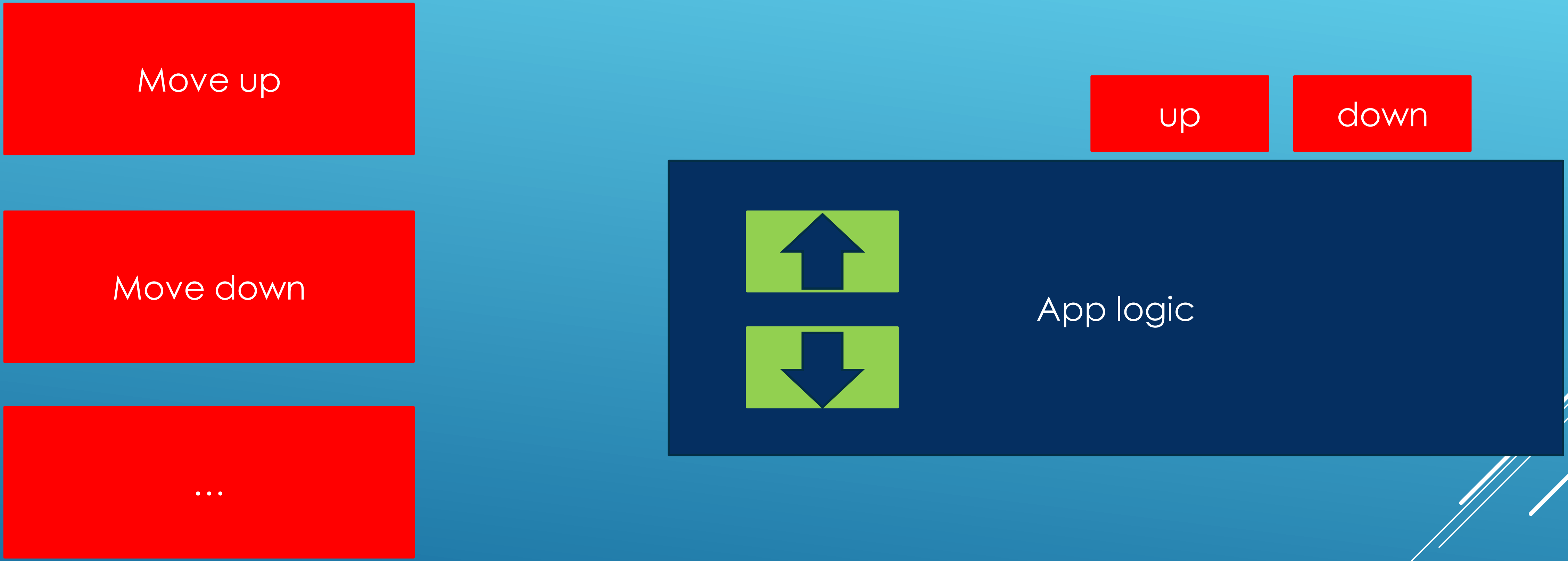
Modify

output

A

The helper function is a callback. It will be called back by the modify function anytime it is needed

Callback functions are heavily used in event based programming like Graphical User Interfaces, but that's out of scope for this course



Move up

Move down

…

up

down

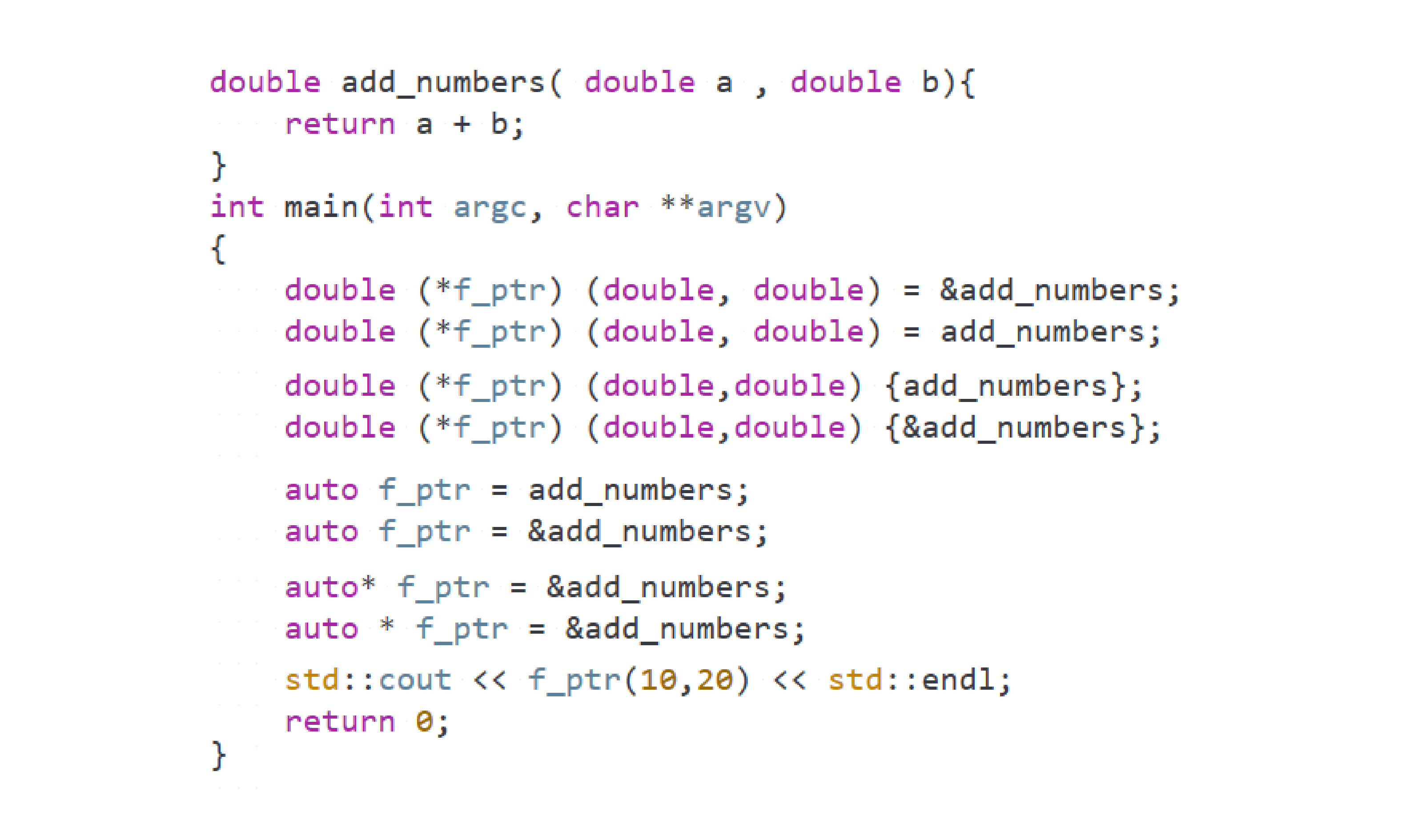
App logic

Slide intentionally left empty

Function pointers

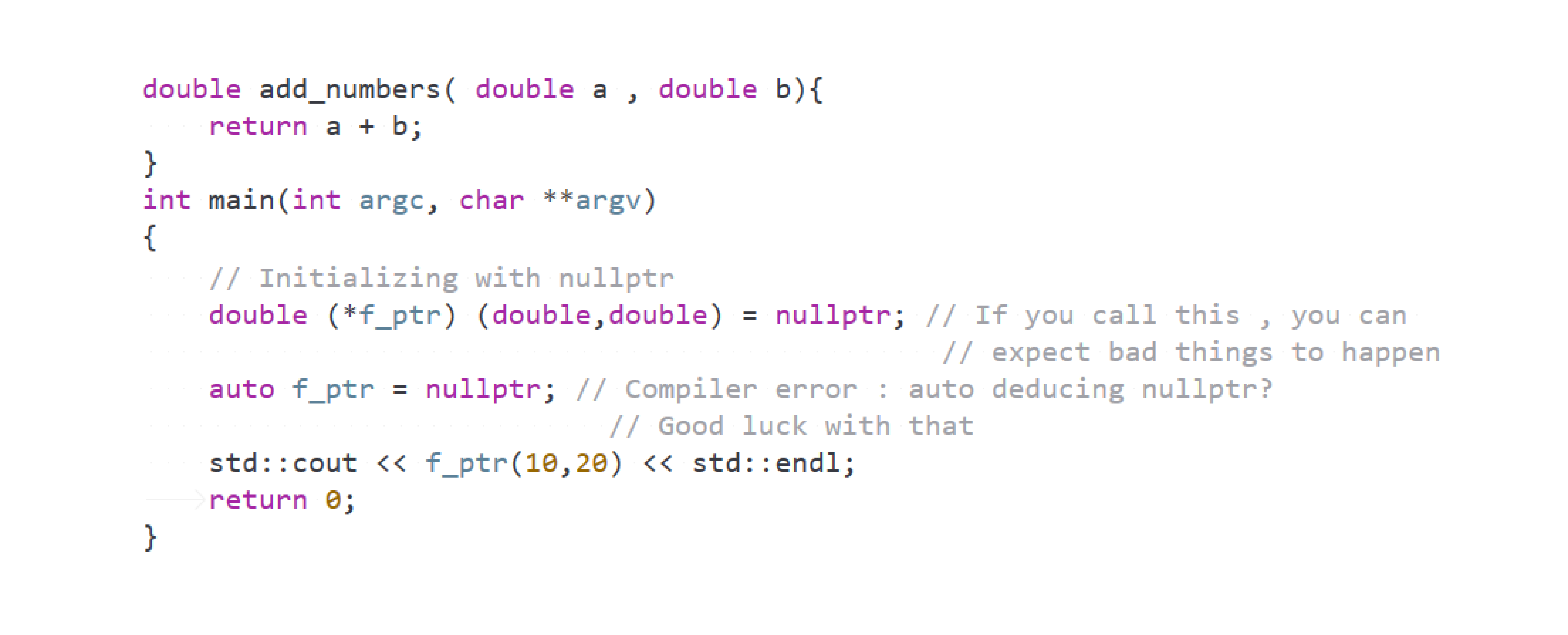
A function is just a block of code that lives somewhere in the memory map of our C++ program. We can grab the address of the function and store it in a function pointer. We'll see how in this lecture

Function pointers : Syntax



```
double add_numbers( double a , double b){
    return a + b;
}
int main(int argc, char **argv)
{
    double (*f_ptr) (double, double) = &add_numbers;
    double (*f_ptr) (double, double) = add_numbers;

    double (*f_ptr) (double,double) {add_numbers};
    double (*f_ptr) (double,double) {&add_numbers};

    auto f_ptr = add_numbers;
    auto f_ptr = &add_numbers;

    auto* f_ptr = &add_numbers;
    auto * f_ptr = &add_numbers;

    std::cout << f_ptr(10,20) << std::endl;
    return 0;
}
```

Initializing with nullptr



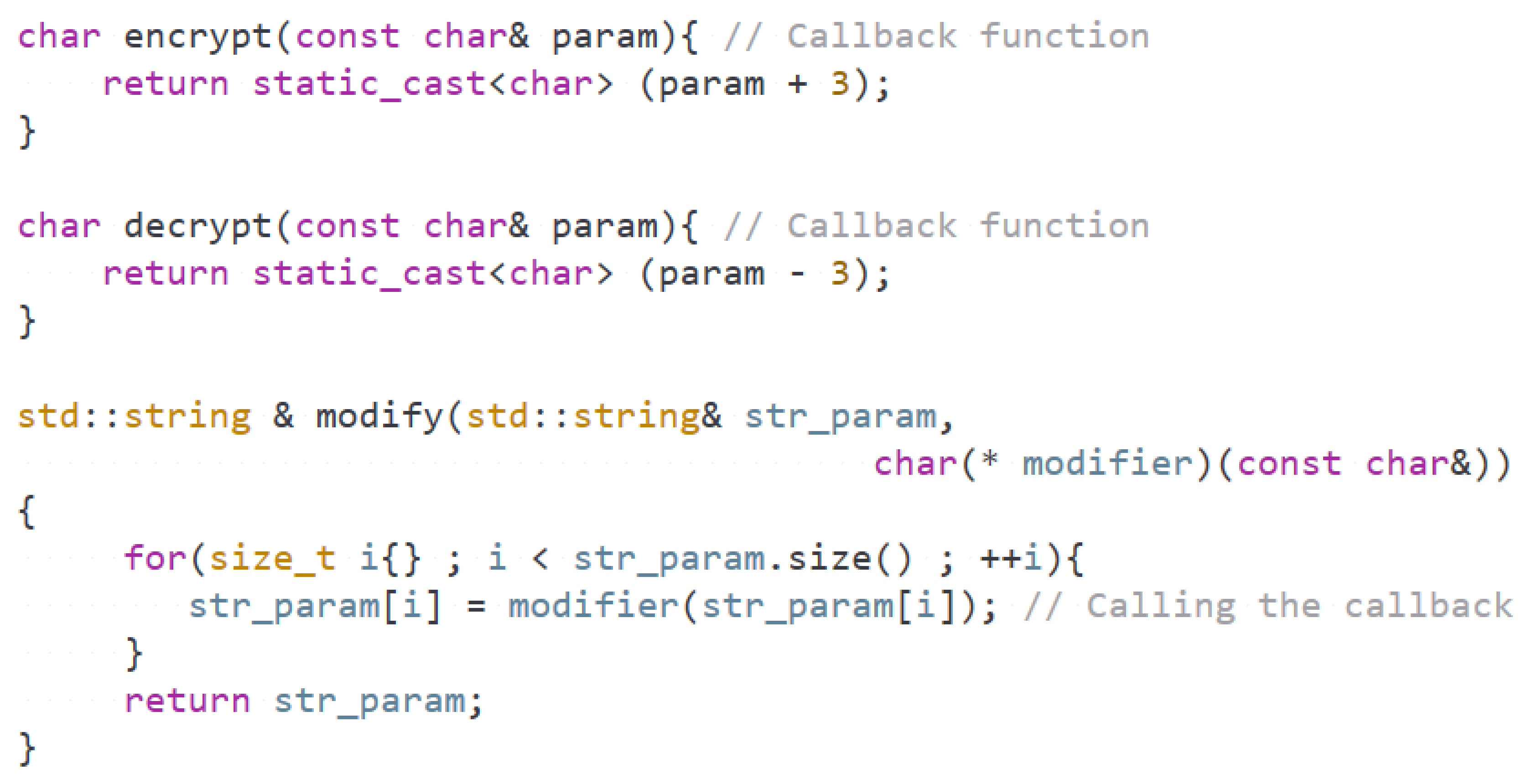
```cpp
double add_numbers( double a , double b){
    return a + b;
}
int main(int argc, char **argv)
{
    // Initializing with nullptr
    double (*f_ptr) (double,double) = nullptr; // If you call this , you can
                                                // expect bad things to happen
    auto f_ptr = nullptr; // Compiler error : auto deducing nullptr?
                            // Good luck with that
    std::cout << f_ptr(10,20) << std::endl;
    return 0;
}
```

Slide intentionally left empty

Callback functions

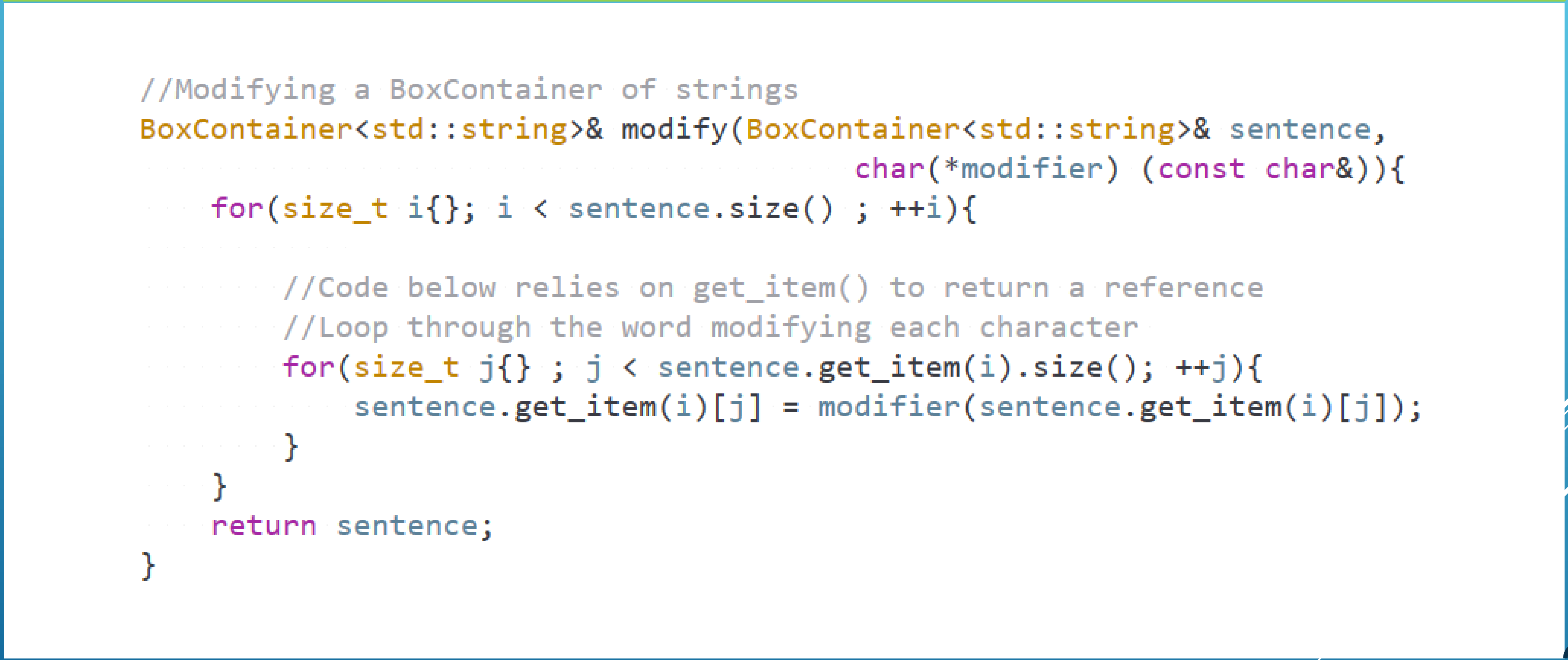
A callback function is a function whose function pointer may be passed to another function as a parameter, and be called somewhere in the body of that function

Callback functions



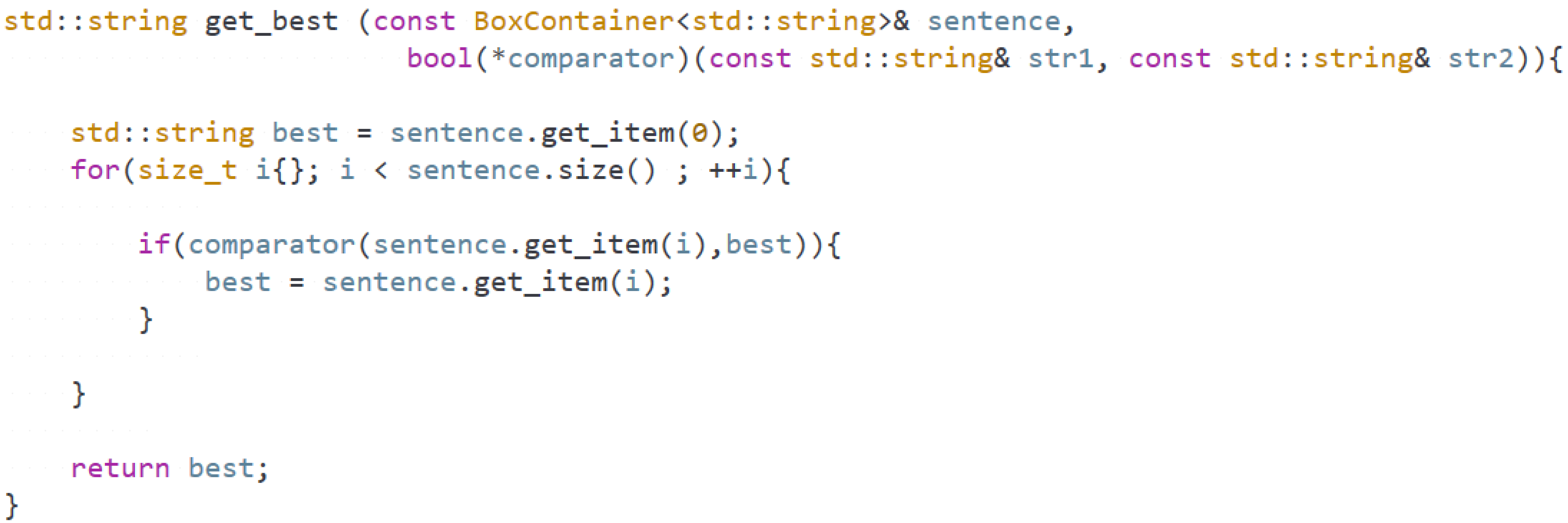
```
char encrypt(const char& param){ // Callback function
    return static_cast<char> (param + 3);
}

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

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;
}
```

A BoxContainer of std::strings



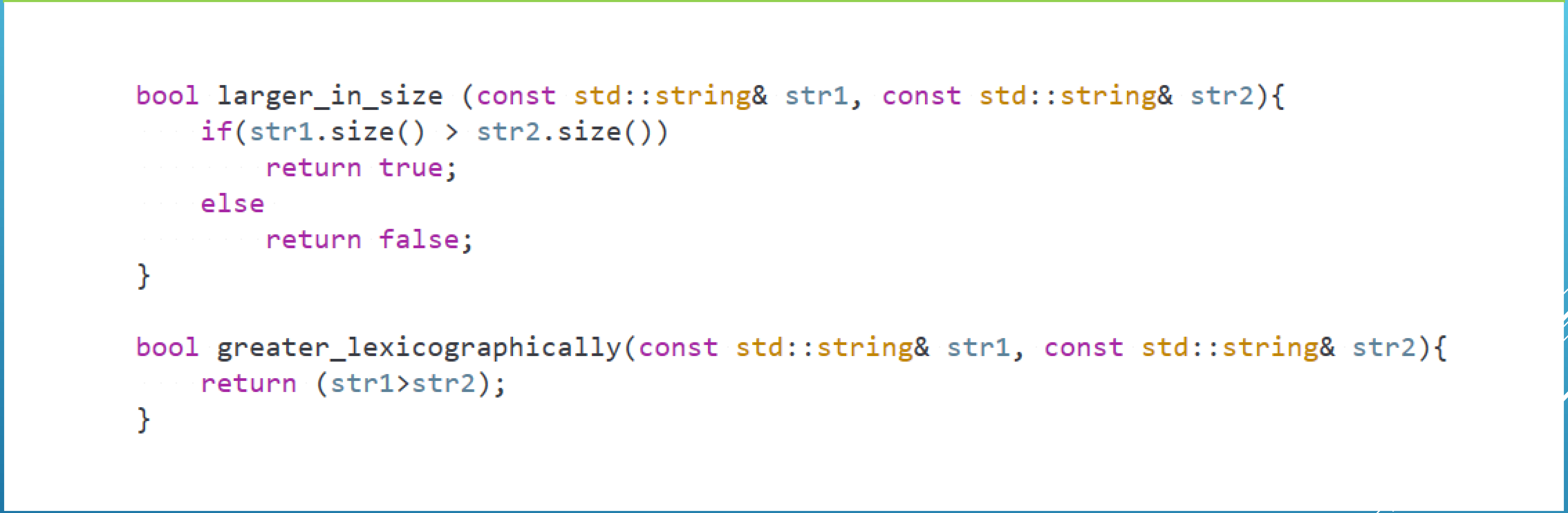
```
//Modifying a BoxContainer of strings
BoxContainer<std::string>& modify(BoxContainer<std::string>& sentence,
                                        char(*modifier) (const char&)){
    for(size_t i{}; i < sentence.size() ; ++i){

        //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;
}
```

BoxContainer : Getting the best string



```
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;
}
```

The definition of best

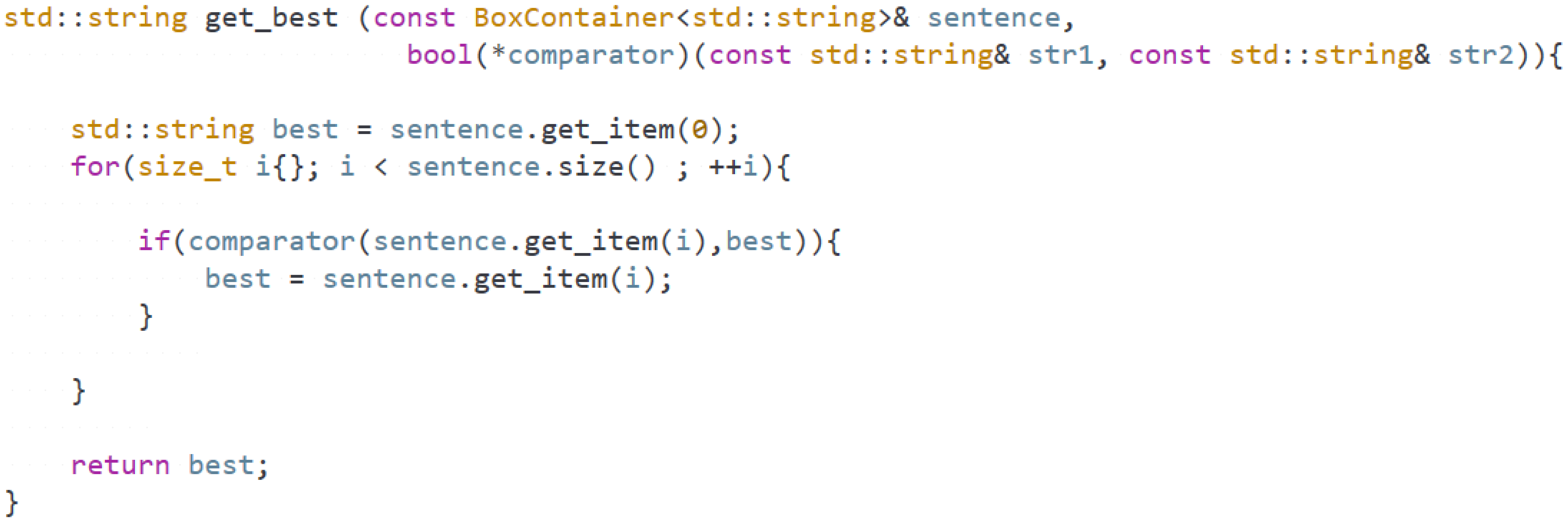


```cpp
bool larger_in_size (const std::string& str1, const std::string& str2){
    if(str1.size() > str2.size())
        return true;
    else
        return false;
}

bool greater_lexicographically(const std::string& str1, const std::string& str2){
    return (str1>str2);
}
```

Slide intentionally left empty

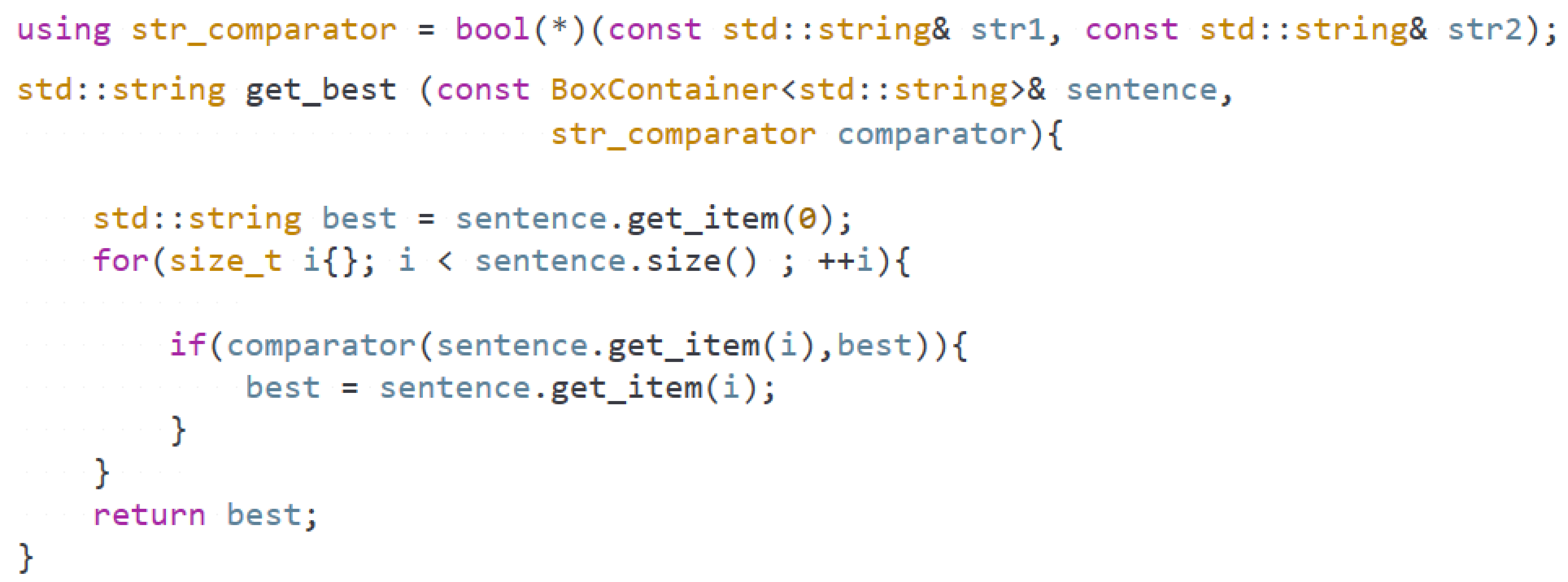
Function pointer type aliases

BoxContainer : Getting the best string

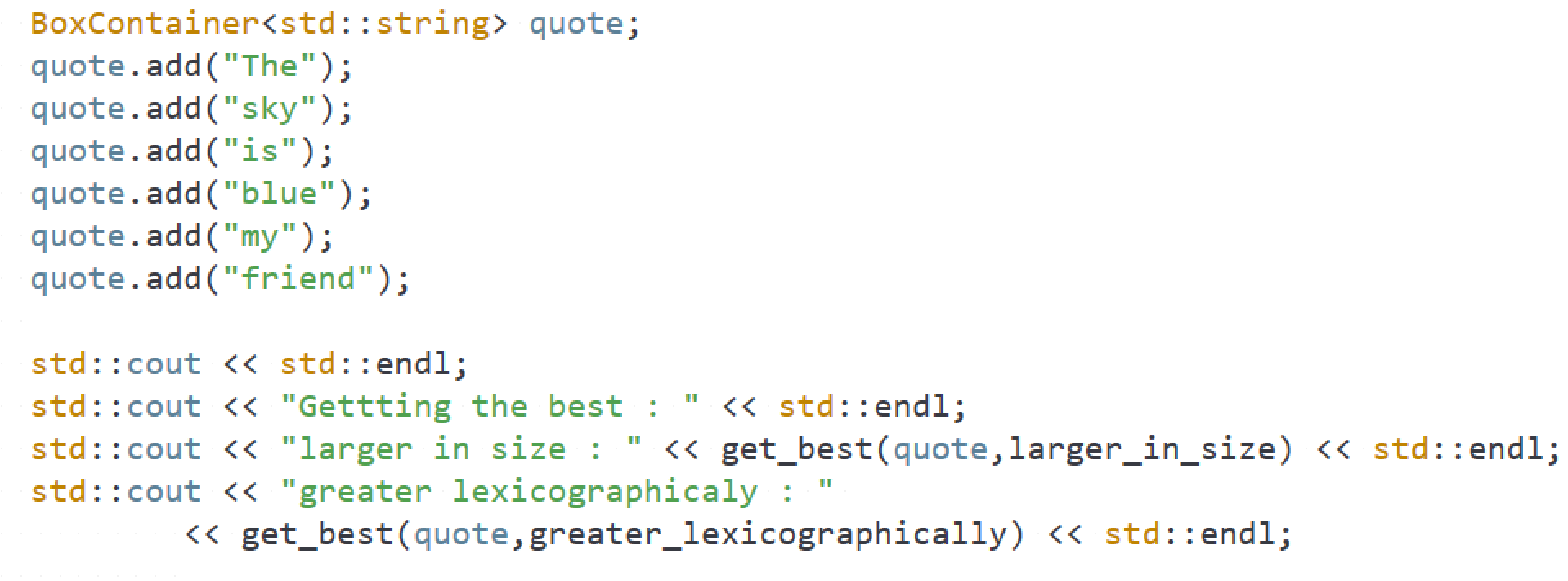


```cpp
std::string get_best (const BoxContainer<std::string>& sentence,
                        bool(*comparator)(const std::string& str1, const std::string& str2)){

    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;
}
```

Type aliases with “using”

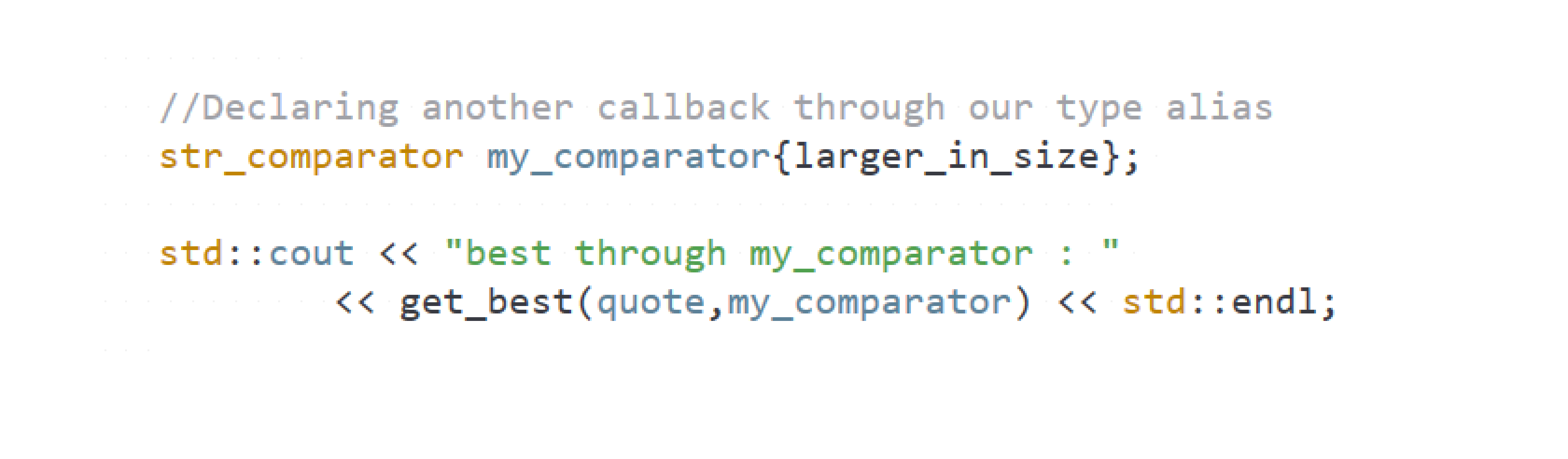


```cpp
using str_comparator = bool(*)(const std::string& str1, const std::string& str2);

std::string get_best (const BoxContainer<std::string>& sentence,
                            str_comparator comparator){

    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;
}
```



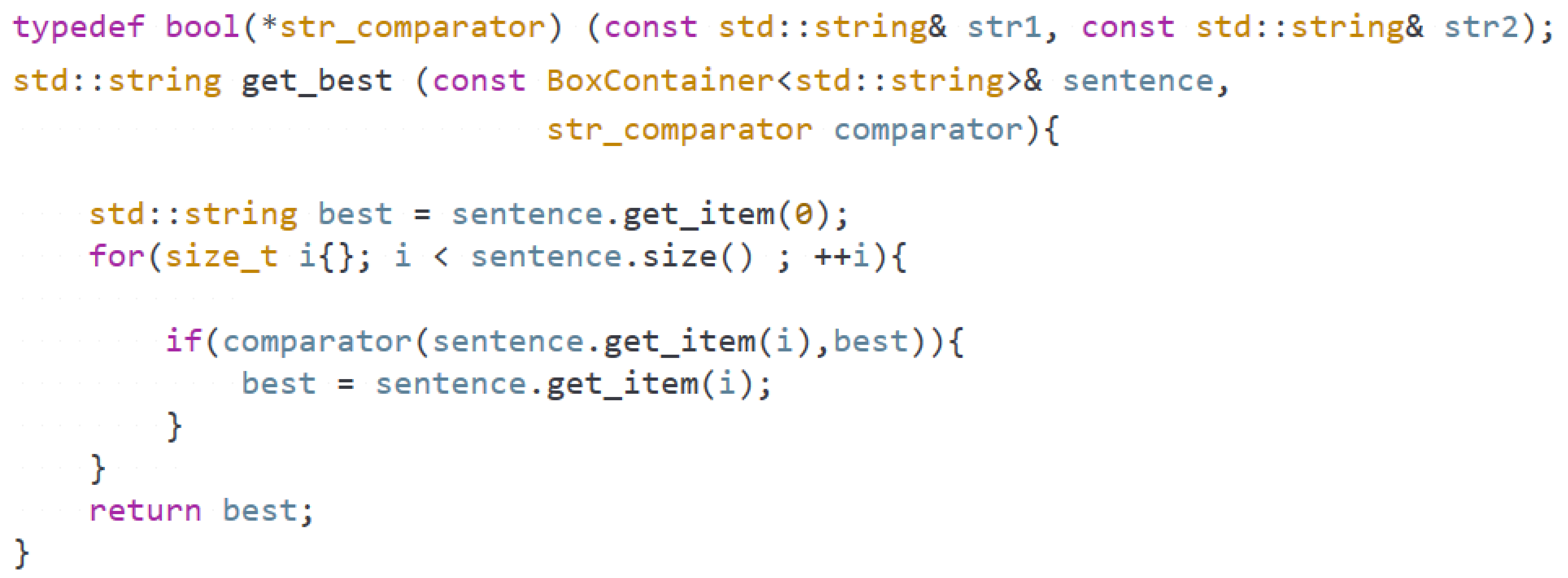
```
BoxContainer<std::string> quote;
quote.add("The");
quote.add("sky");
quote.add("is");
quote.add("blue");
quote.add("my");
quote.add("friend");

std::cout << std::endl;
std::cout << "Gettting the best : " << std::endl;
std::cout << "larger in size : " << get_best(quote,larger_in_size) << std::endl;
std::cout << "greater lexicographicaly : "
        << get_best(quote,greater_lexicographically) << std::endl;
```

Declaring new callbacks with type aliases



```cpp
//Declaring another callback through our type alias
str_comparator my_comparator{larger_in_size};

std::cout << "best through my_comparator : "
        << get_best(quote,my_comparator) << std::endl;
```

typedef syntax

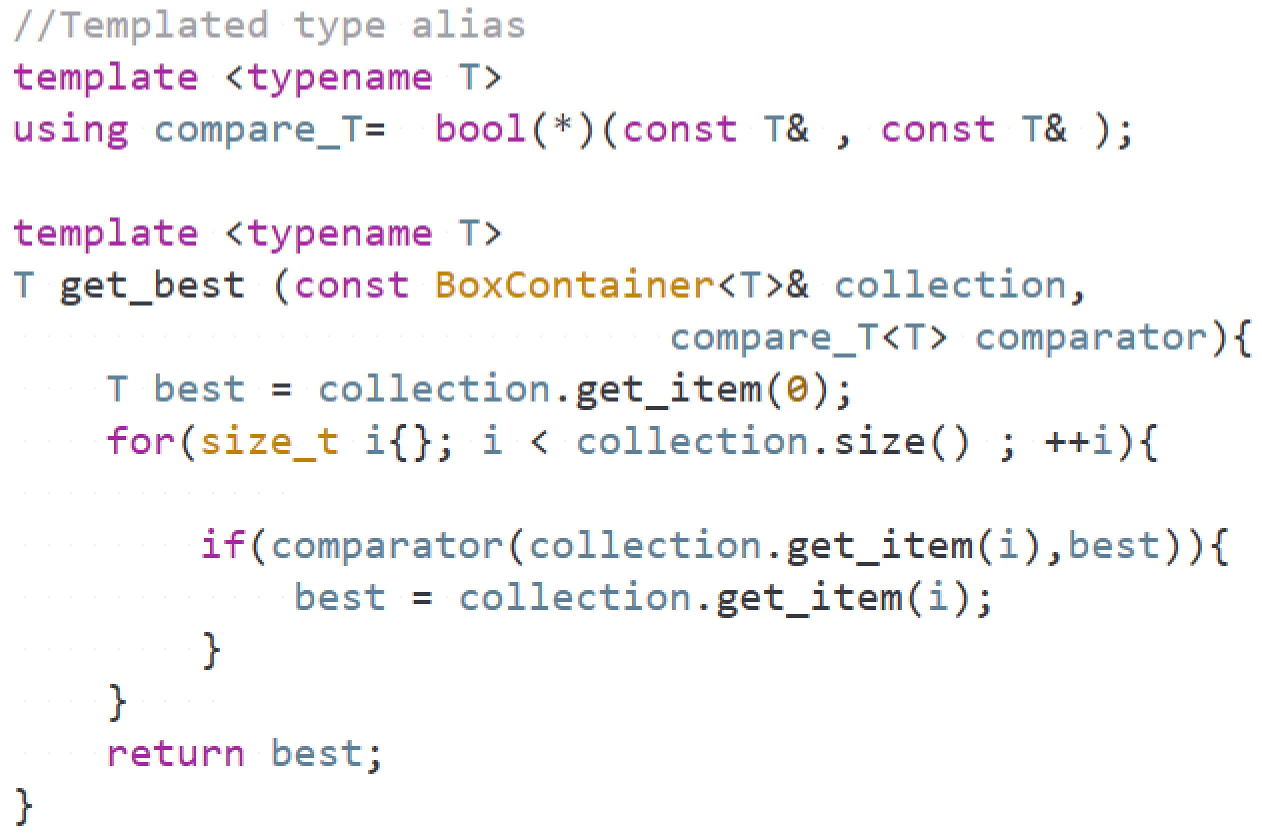


```cpp
typedef bool(*str_comparator) (const std::string& str1, const std::string& str2);
std::string get_best (const BoxContainer<std::string>& sentence,
                            str_comparator comparator){

    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;
}
```

Slide intentionally left empty

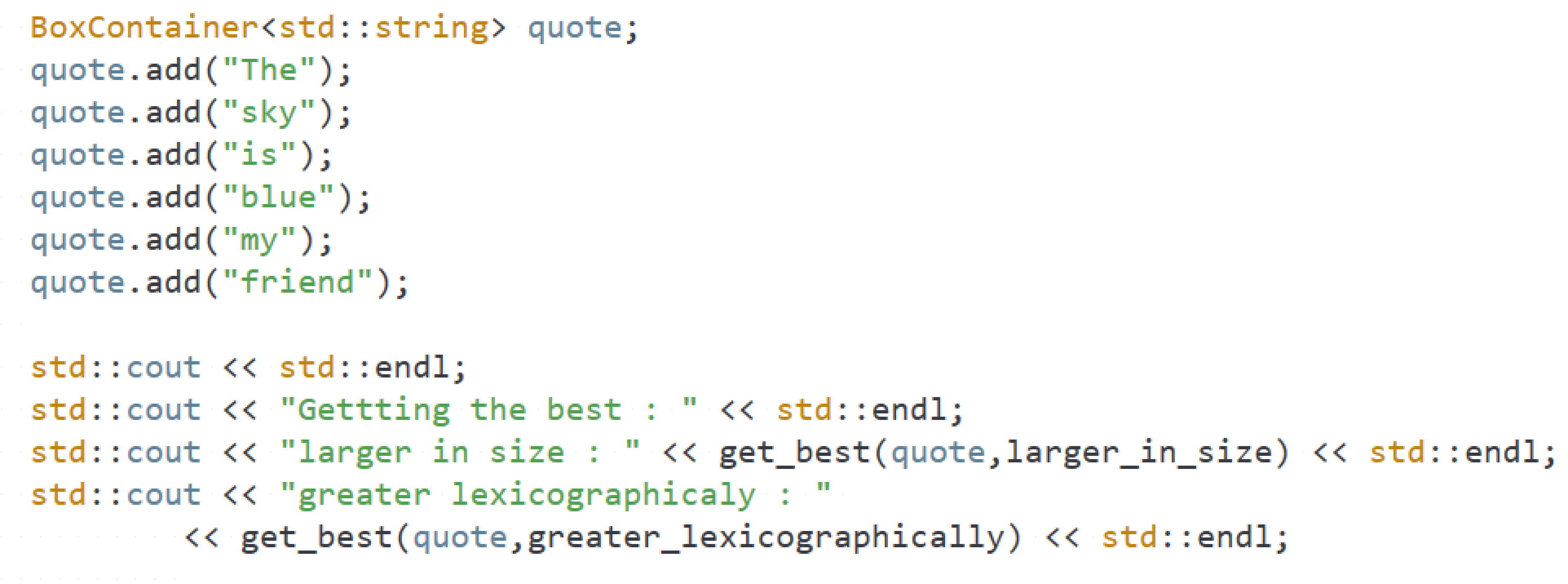
Function pointer type aliases with templates

Type alias with templates



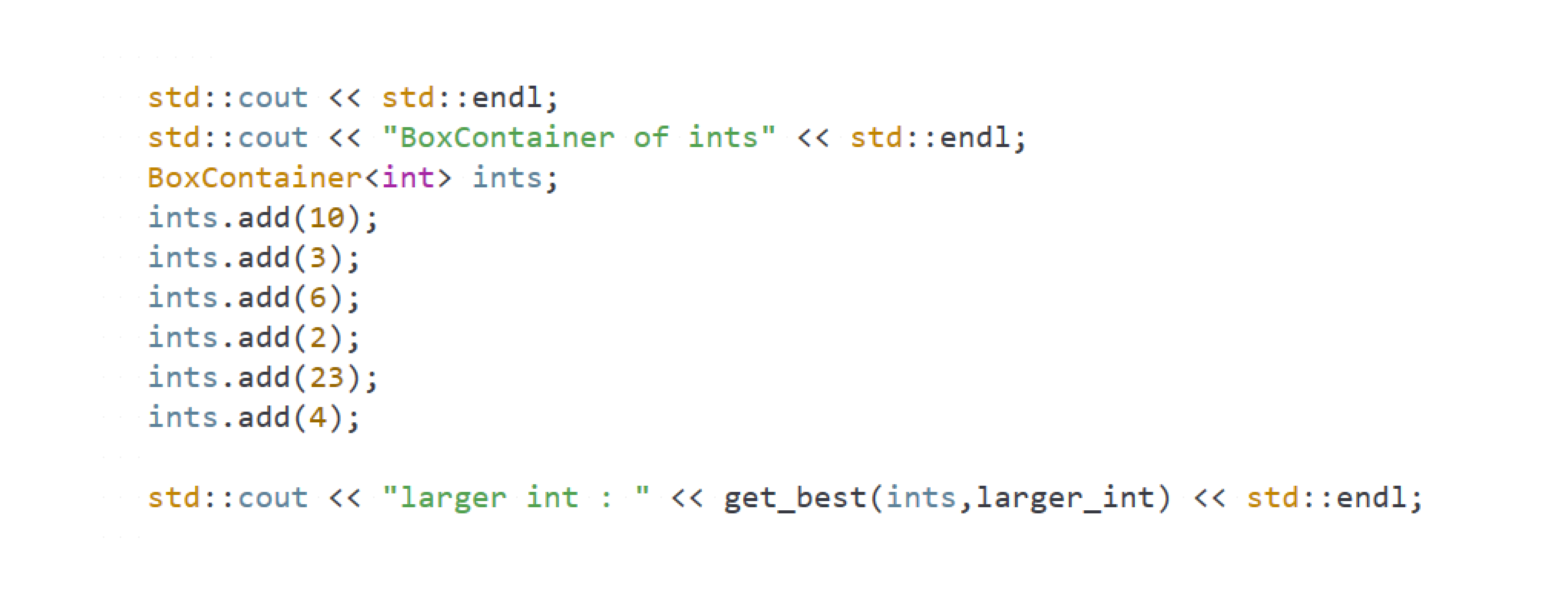
```cpp
//Templated type alias
template <typename T>
using compare_T=  bool(*)(const T& , const T& );

template <typename T>
T get_best (const BoxContainer<T>& collection,
                            compare_T<T> comparator){
    T best = collection.get_item(0);
    for(size_t i{}; i < collection.size() ; ++i){

        if(comparator(collection.get_item(i),best)){
            best = collection.get_item(i);
        }
    }
    return best;
}
```

Std::strings stored in BoxContainer



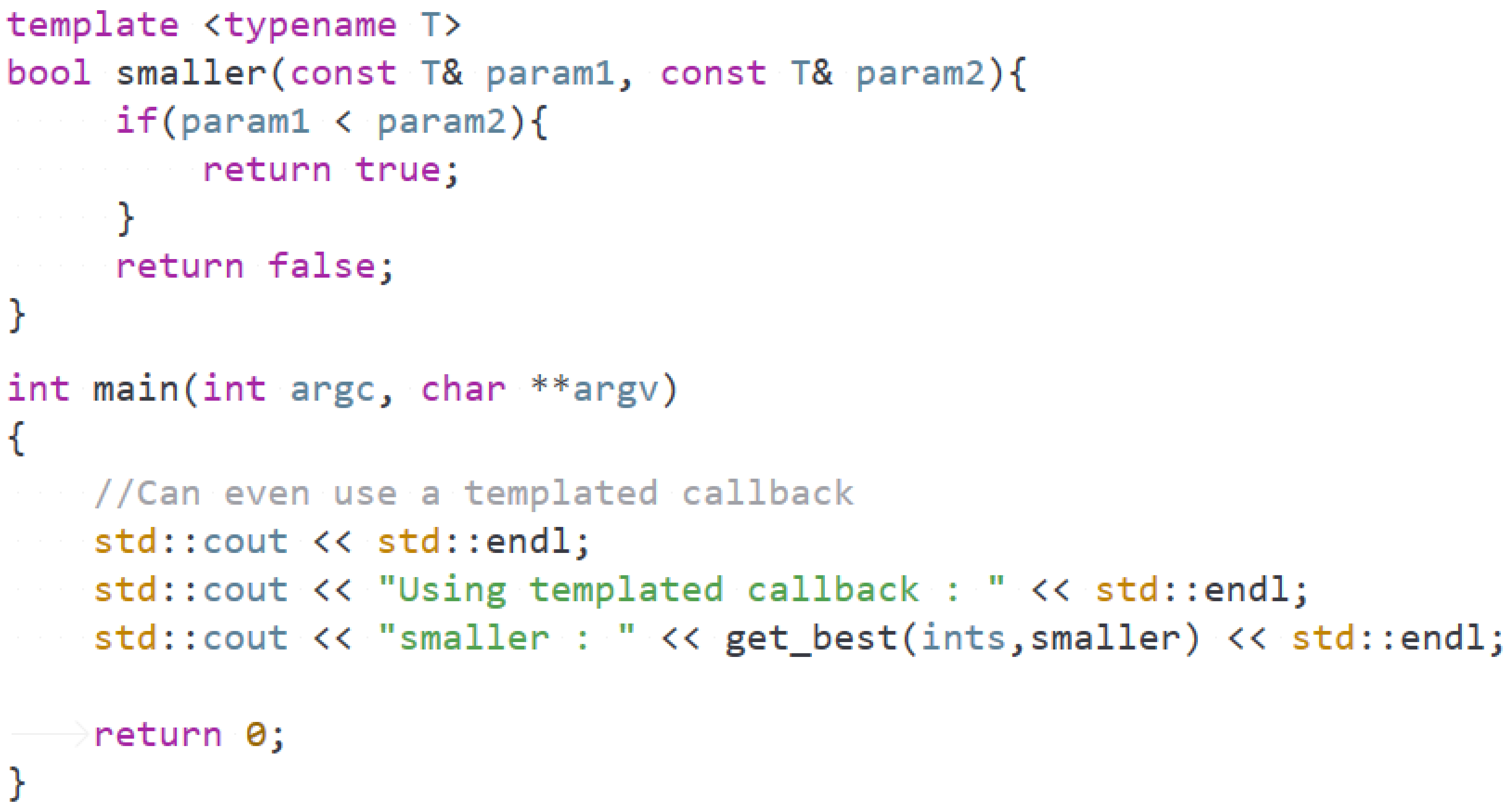
```cpp
BoxContainer<std::string> quote;
quote.add("The");
quote.add("sky");
quote.add("is");
quote.add("blue");
quote.add("my");
quote.add("friend");

std::cout << std::endl;
std::cout << "Gettting the best : " << std::endl;
std::cout << "larger in size : " << get_best(quote,larger_in_size) << std::endl;
std::cout << "greater lexicographicaly : "
        << get_best(quote,greater_lexicographically) << std::endl;
```

ints stored in BoxContainer



```
    std::cout << std::endl;
    std::cout << "BoxContainer of ints" << std::endl;
    BoxContainer<int> ints;
    ints.add(10);
    ints.add(3);
    ints.add(6);
    ints.add(2);
    ints.add(23);
    ints.add(4);

    std::cout << "larger int : " << get_best(ints,larger_int) << std::endl;
```

Templated callback



```
template <typename T>
bool smaller(const T& param1, const T& param2){
    if(param1 < param2){
        return true;
    }
    return false;
}

int main(int argc, char **argv)
{
    //Can even use a templated callback
    std::cout << std::endl;
    std::cout << "Using templated callback : " << std::endl;
    std::cout << "smaller : " << get_best(ints,smaller) << std::endl;

    return 0;
}
```

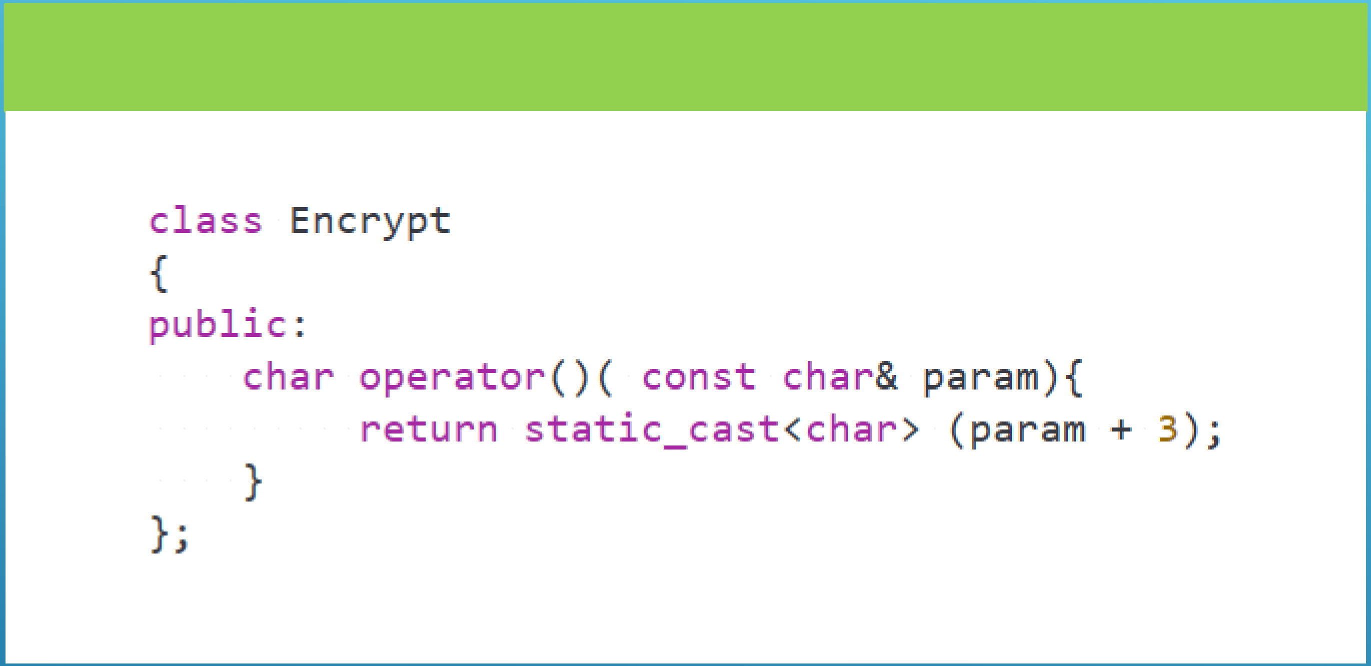
typedef syntax

Templated type aliases for function pointers don’t work with the typedef syntax. This is another reason to avoid them in new modern C++ code

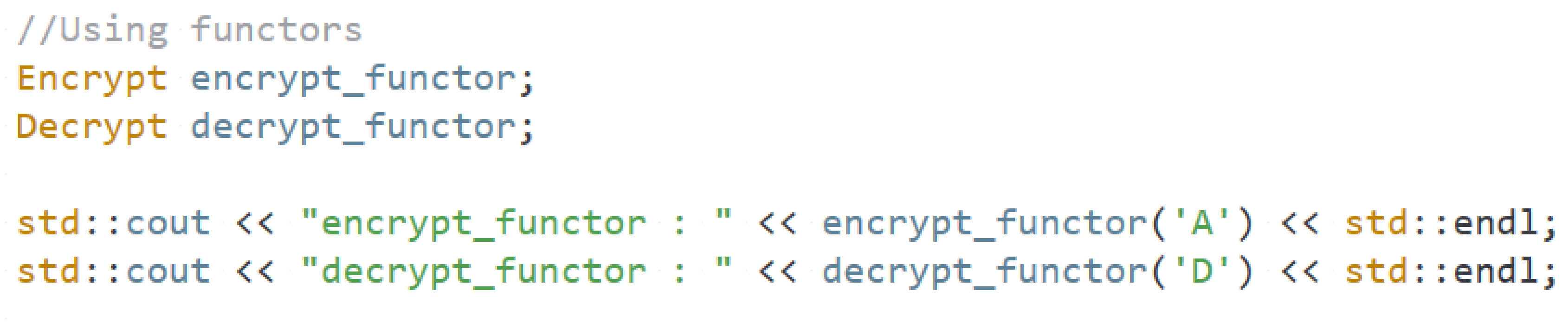
Slide intentionally left empty

Functors

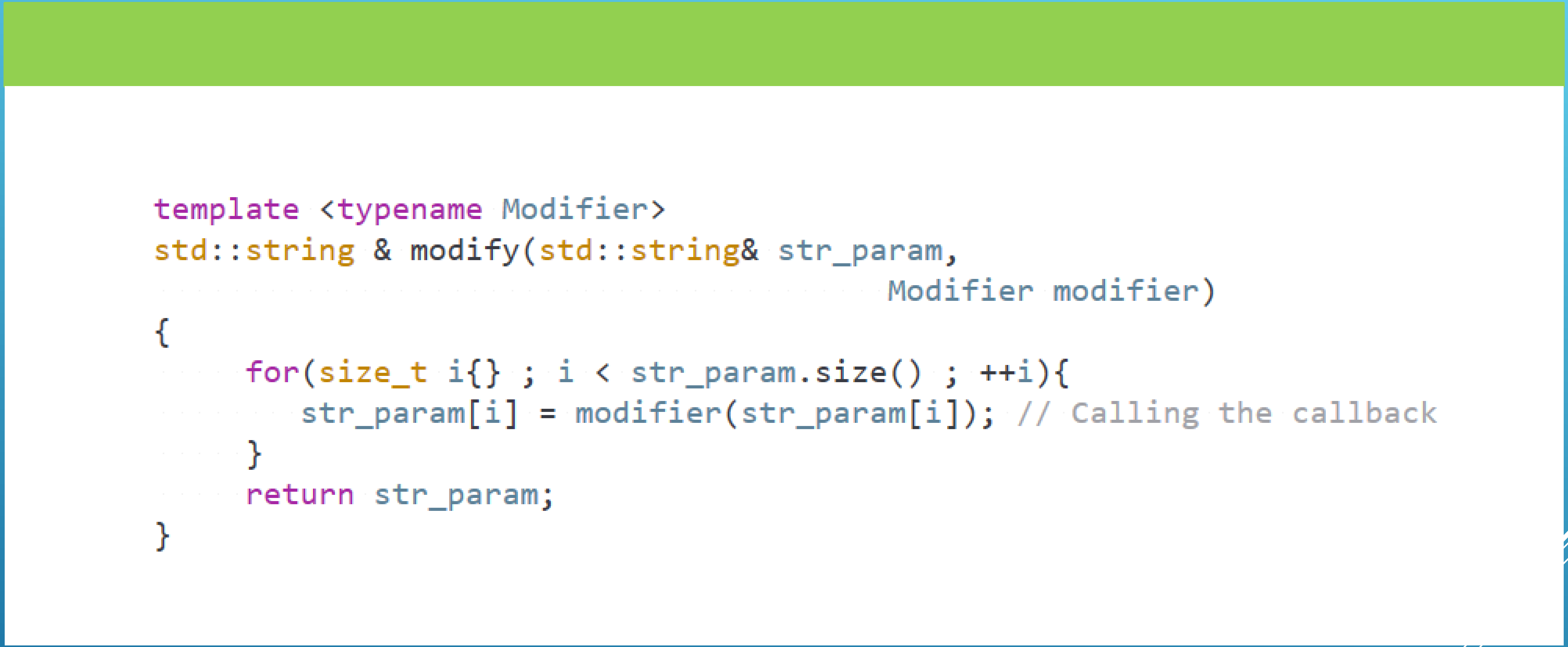
- Class objects that can be called like ordinary functions
- We set them up by overloading the () operator for our class



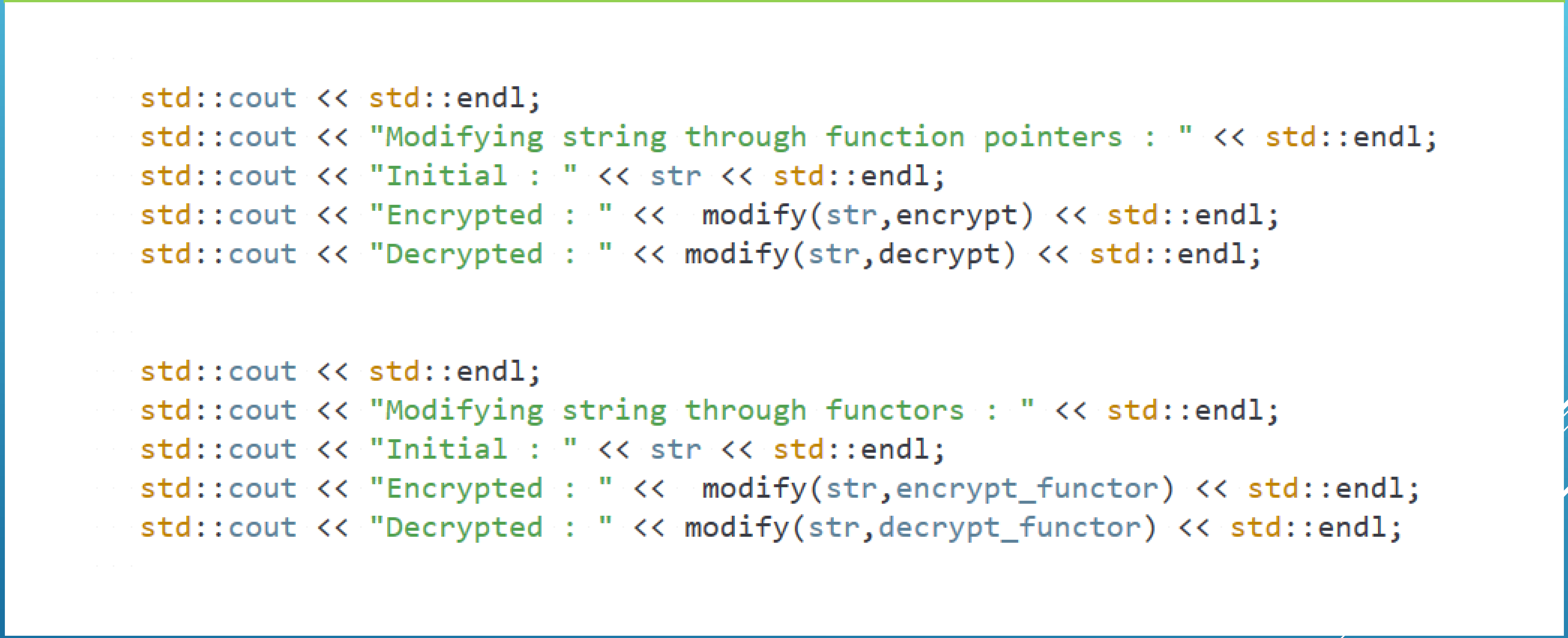
```
class Encrypt
{
public:
    char operator()( const char& param){
         return static_cast<char> (param + 3);
    }
};
```



```
//Using functors
Encrypt encrypt_functor;
Decrypt decrypt_functor;

std::cout << "encrypt_functor : " << encrypt_functor('A') << std::endl;
std::cout << "decrypt_functor : " << decrypt_functor('D') << std::endl;
```



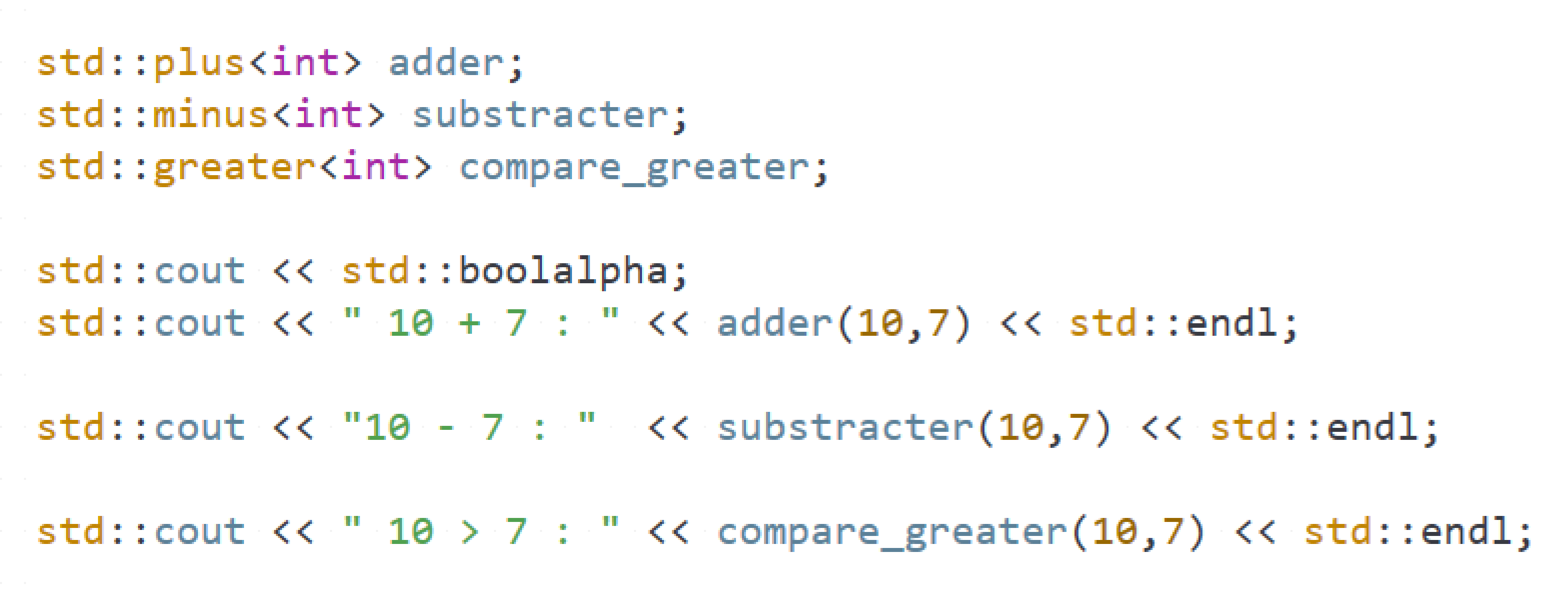
```cpp
template <typename Modifier>
std::string & modify(std::string& str_param,
                                        Modifier modifier)
{
    for(size_t i{} ; i < str_param.size() ; ++i){
        str_param[i] = modifier(str_param[i]); // Calling the callback
    }
    return str_param;
}
```



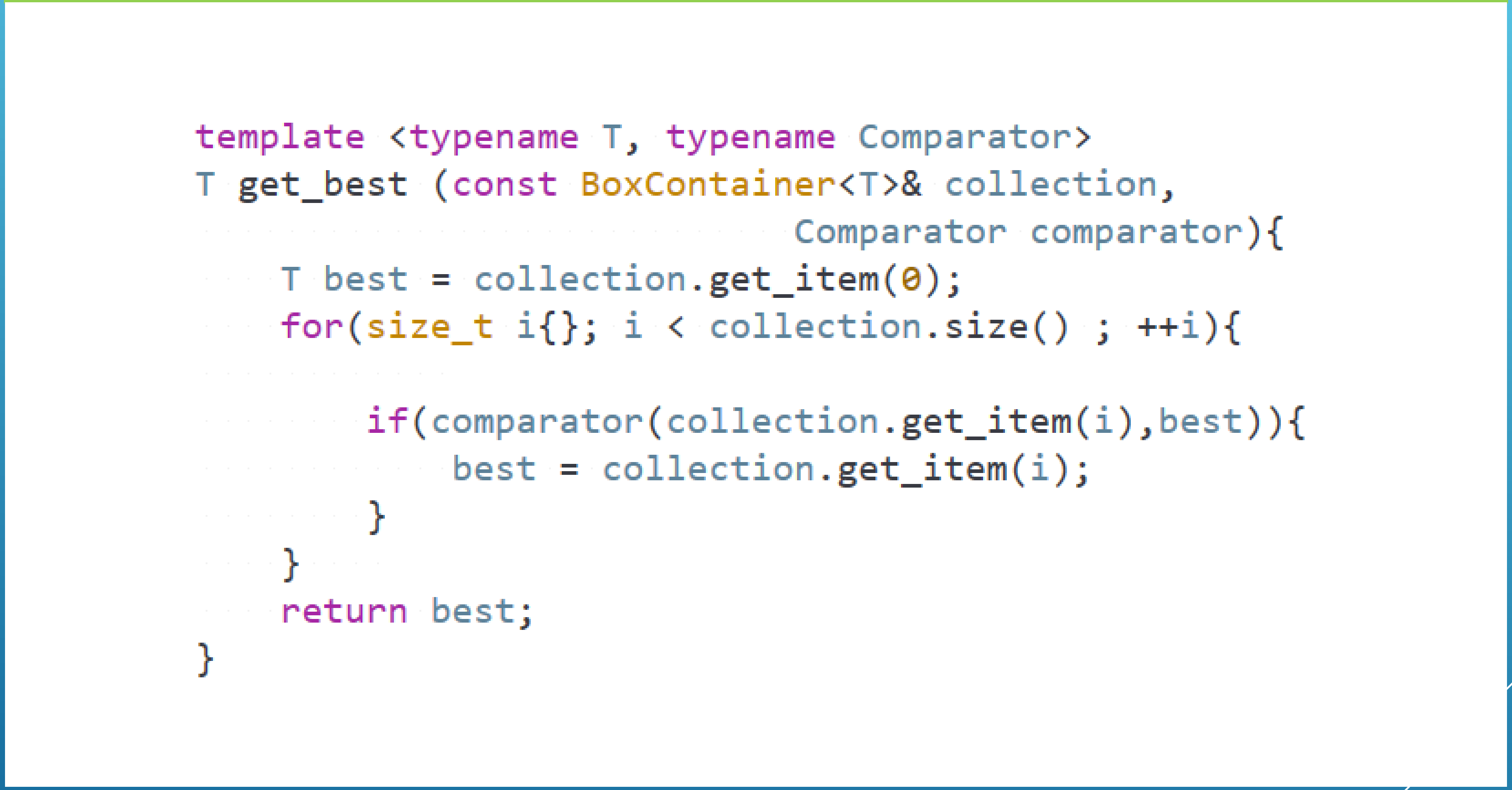
```
    std::cout << std::endl;
    std::cout << "Modifying string through function pointers : " << std::endl;
    std::cout << "Initial : " << str << std::endl;
    std::cout << "Encrypted : " <<  modify(str,encrypt) << std::endl;
    std::cout << "Decrypted : " << modify(str,decrypt) << std::endl;


    std::cout << std::endl;
    std::cout << "Modifying string through functors : " << std::endl;
    std::cout << "Initial : " << str << std::endl;
    std::cout << "Encrypted : " <<  modify(str,encrypt_functor) << std::endl;
    std::cout << "Decrypted : " << modify(str,decrypt_functor) << std::endl;
```

Slide intentionally left empty

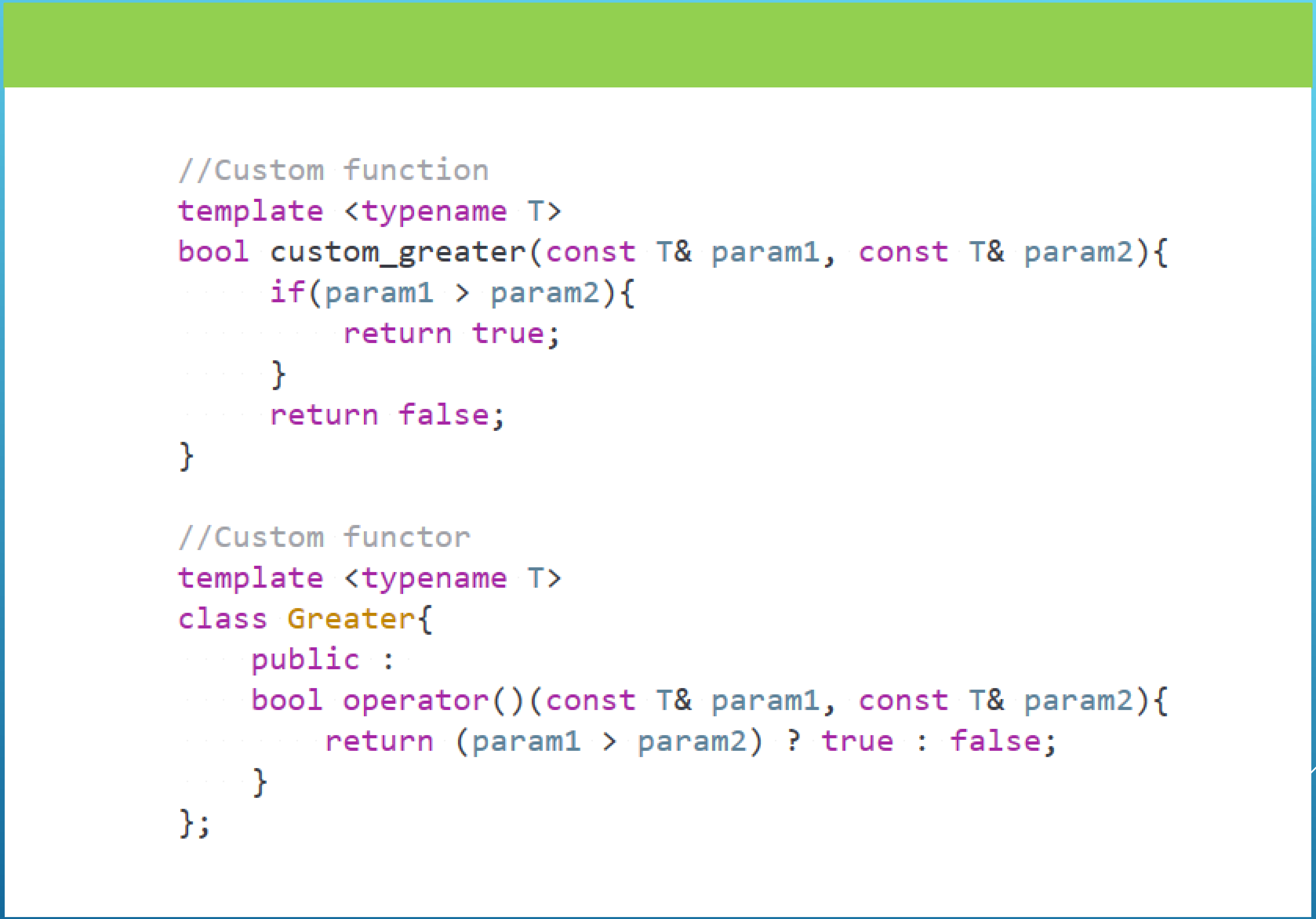
Functors in <functional> header



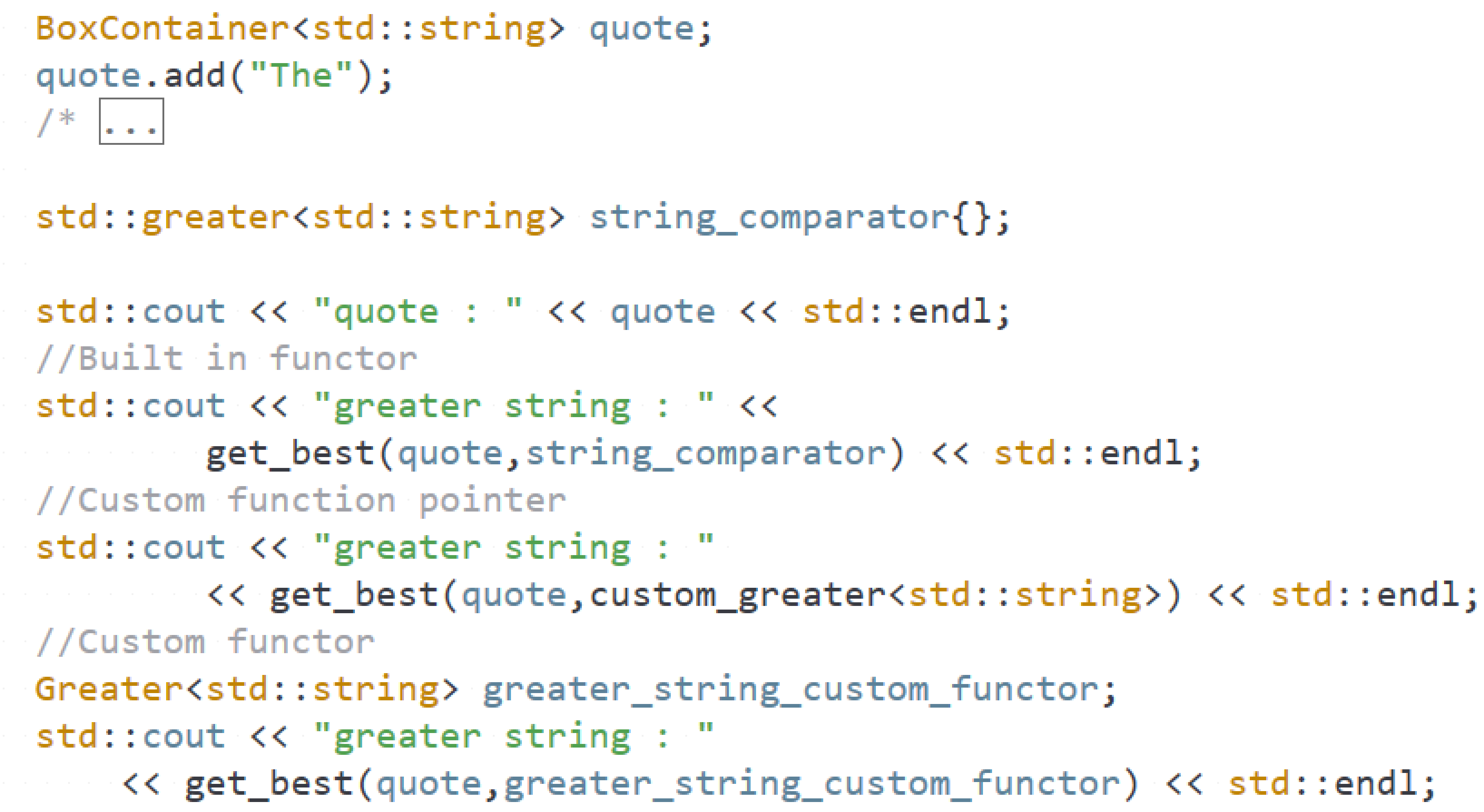
```
std::plus<int> adder;
std::minus<int> substracter;
std::greater<int> compare_greater;

std::cout << std::boolalpha;
std::cout << " 10 + 7 : " << adder(10,7) << std::endl;

std::cout << "10 - 7 : "  << substracter(10,7) << std::endl;

std::cout << " 10 > 7 : " << compare_greater(10,7) << std::endl;
```



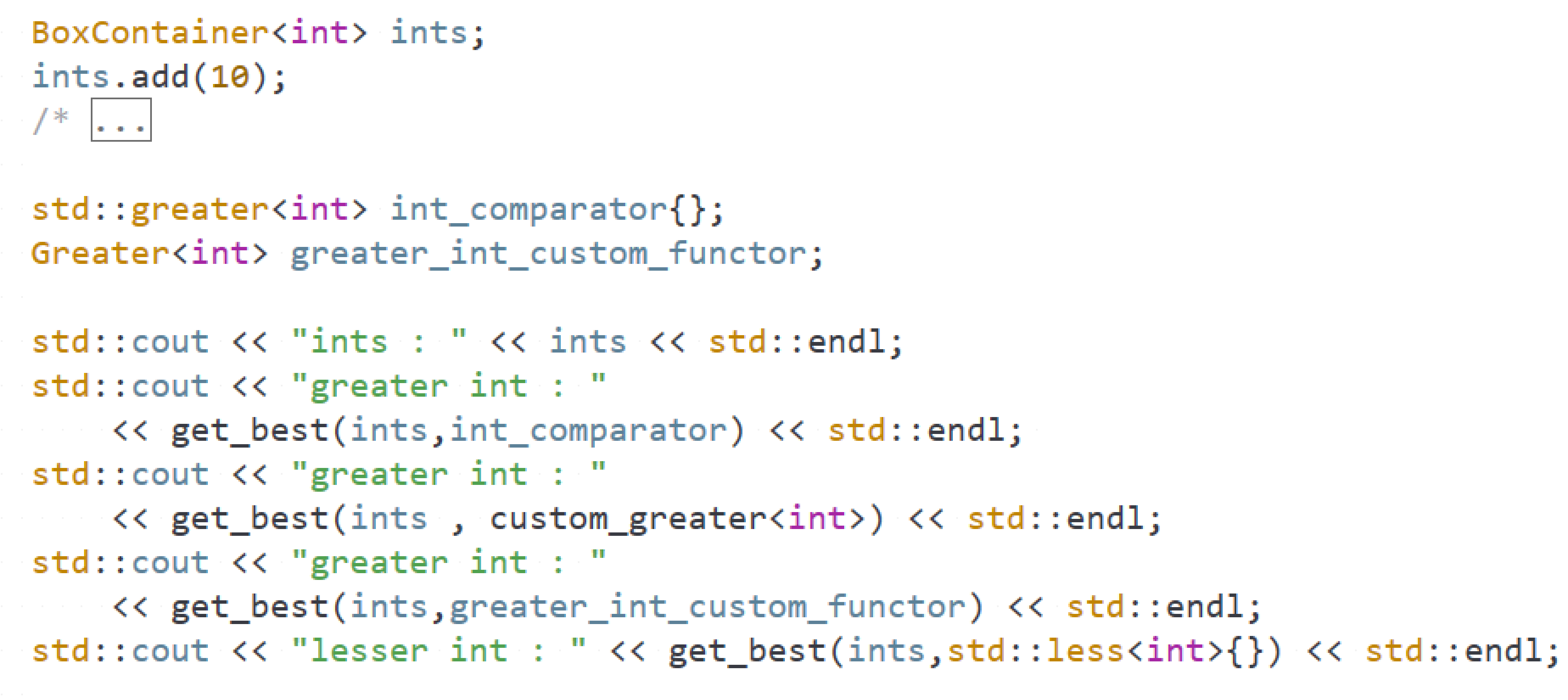
```cpp
template <typename T, typename Comparator>
T get_best (const BoxContainer<T>& collection,
                            Comparator comparator){
    T best = collection.get_item(0);
    for(size_t i{}; i < collection.size() ; ++i){

        if(comparator(collection.get_item(i),best)){
            best = collection.get_item(i);
        }
    }
    return best;
}
```



```
//Custom function
template <typename T>
bool custom_greater(const T& param1, const T& param2){
     if(param1 > param2){
         return true;
     }
     return false;
}

//Custom functor
template <typename T>
class Greater{
    public :
    bool operator()(const T& param1, const T& param2){
        return (param1 > param2) ? true : false;
    }
};
```



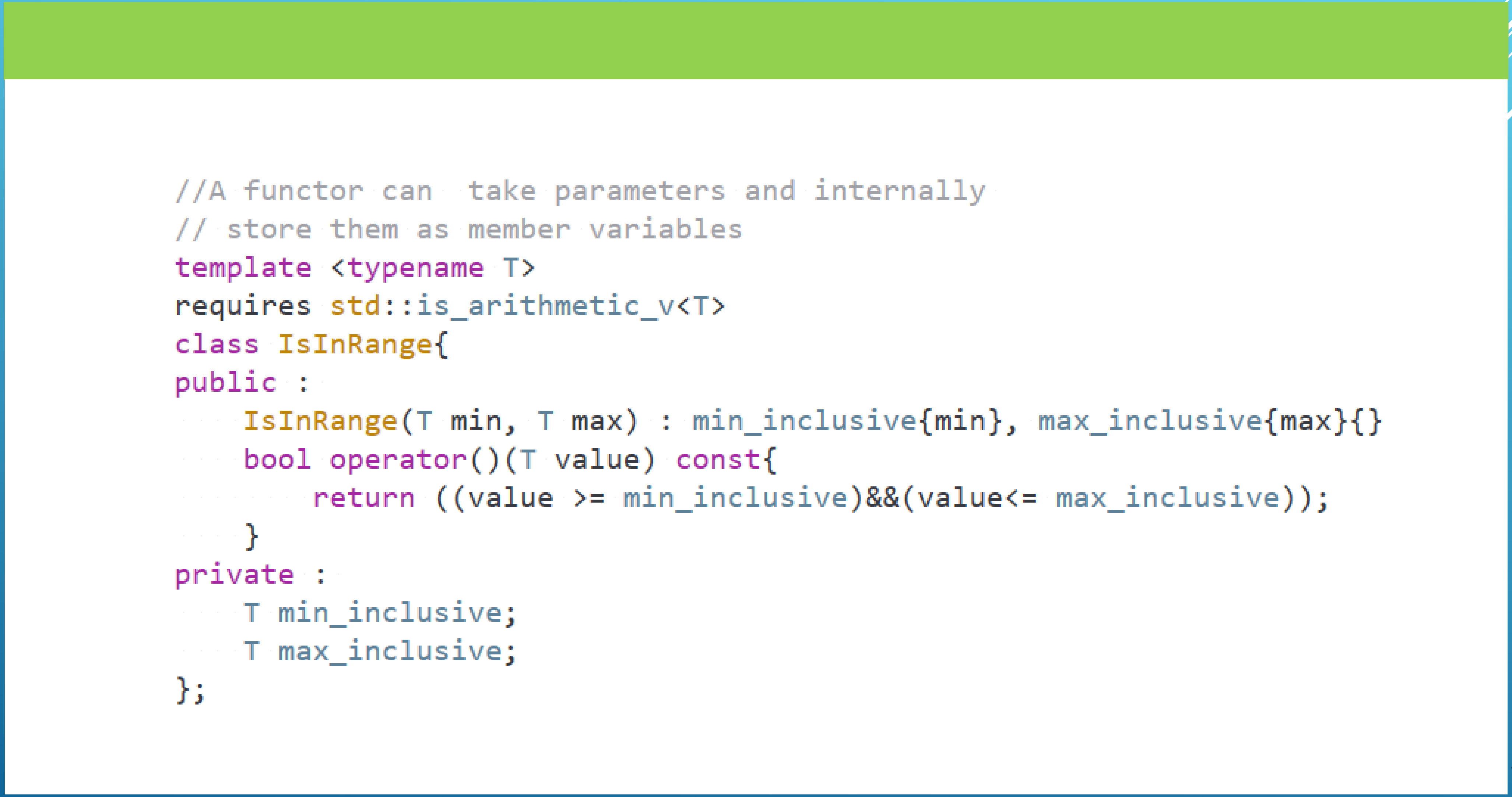
```
BoxContainer<std::string> quote;
quote.add("The");
/* ...

std::greater<std::string> string_comparator{};

std::cout << "quote : " << quote << std::endl;
//Built in functor
std::cout << "greater string : " <<
        get_best(quote,string_comparator) << std::endl;
//Custom function pointer
std::cout << "greater string : "
        << get_best(quote,custom_greater<std::string>) << std::endl;
//Custom functor
Greater<std::string> greater_string_custom_functor;
std::cout << "greater string : "
    << get_best(quote,greater_string_custom_functor) << std::endl;
```



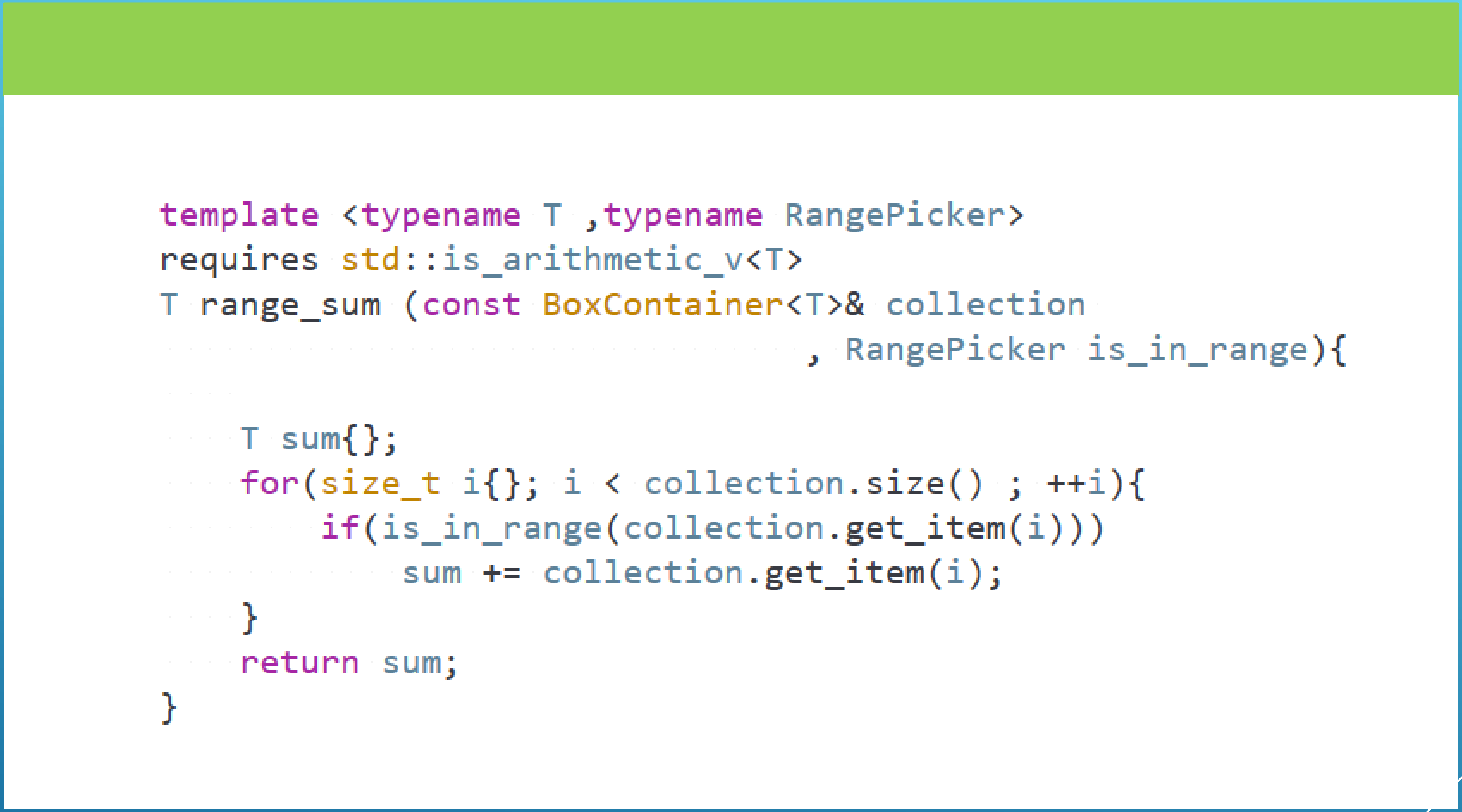
```
BoxContainer<int> ints;
ints.add(10);
/* ...

std::greater<int> int_comparator{};
Greater<int> greater_int_custom_functor;

std::cout << "ints : " << ints << std::endl;
std::cout << "greater int : "
    << get_best(ints,int_comparator) << std::endl;
std::cout << "greater int : "
    << get_best(ints , custom_greater<int>) << std::endl;
std::cout << "greater int : "
    << get_best(ints,greater_int_custom_functor) << std::endl;
std::cout << "lesser int : " << get_best(ints,std::less<int>{}) << std::endl;
```

Slide intentionally left empty

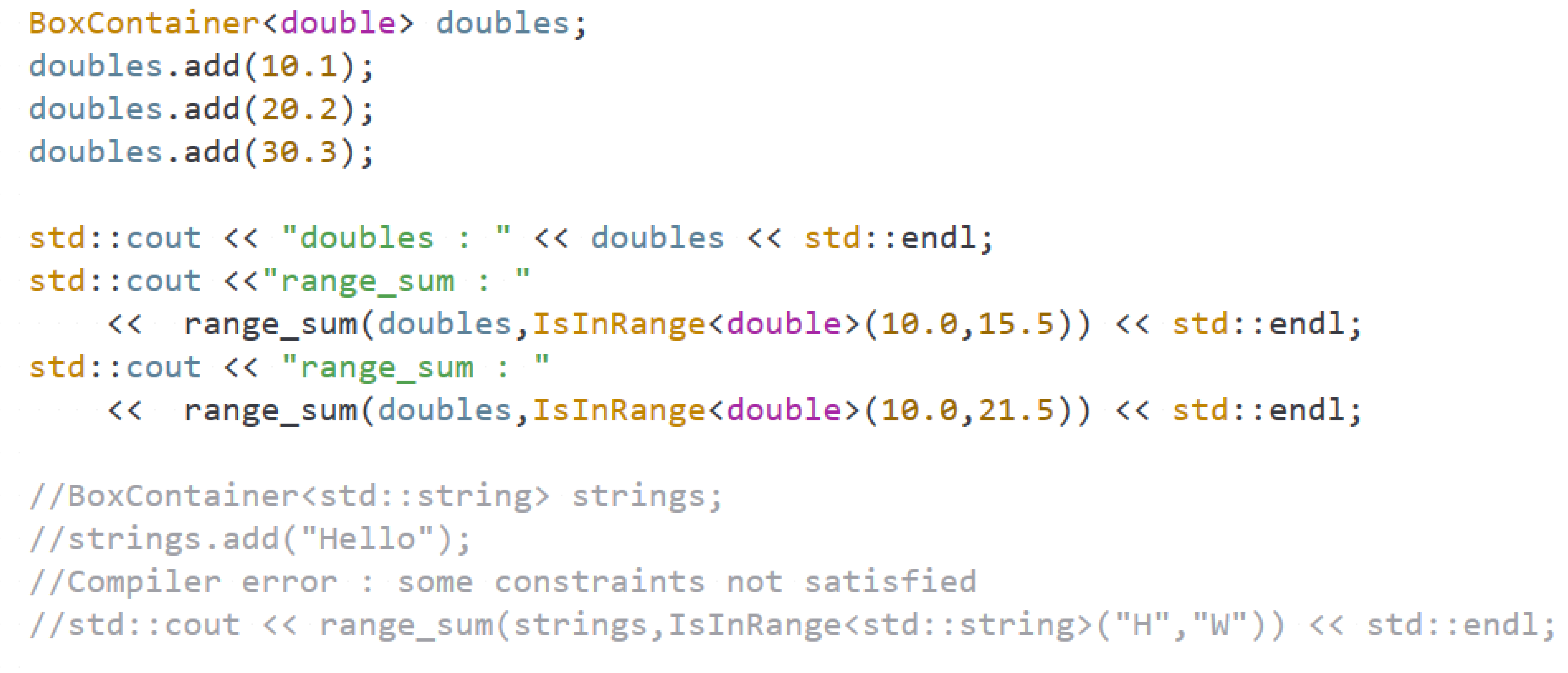
Functors with parameters



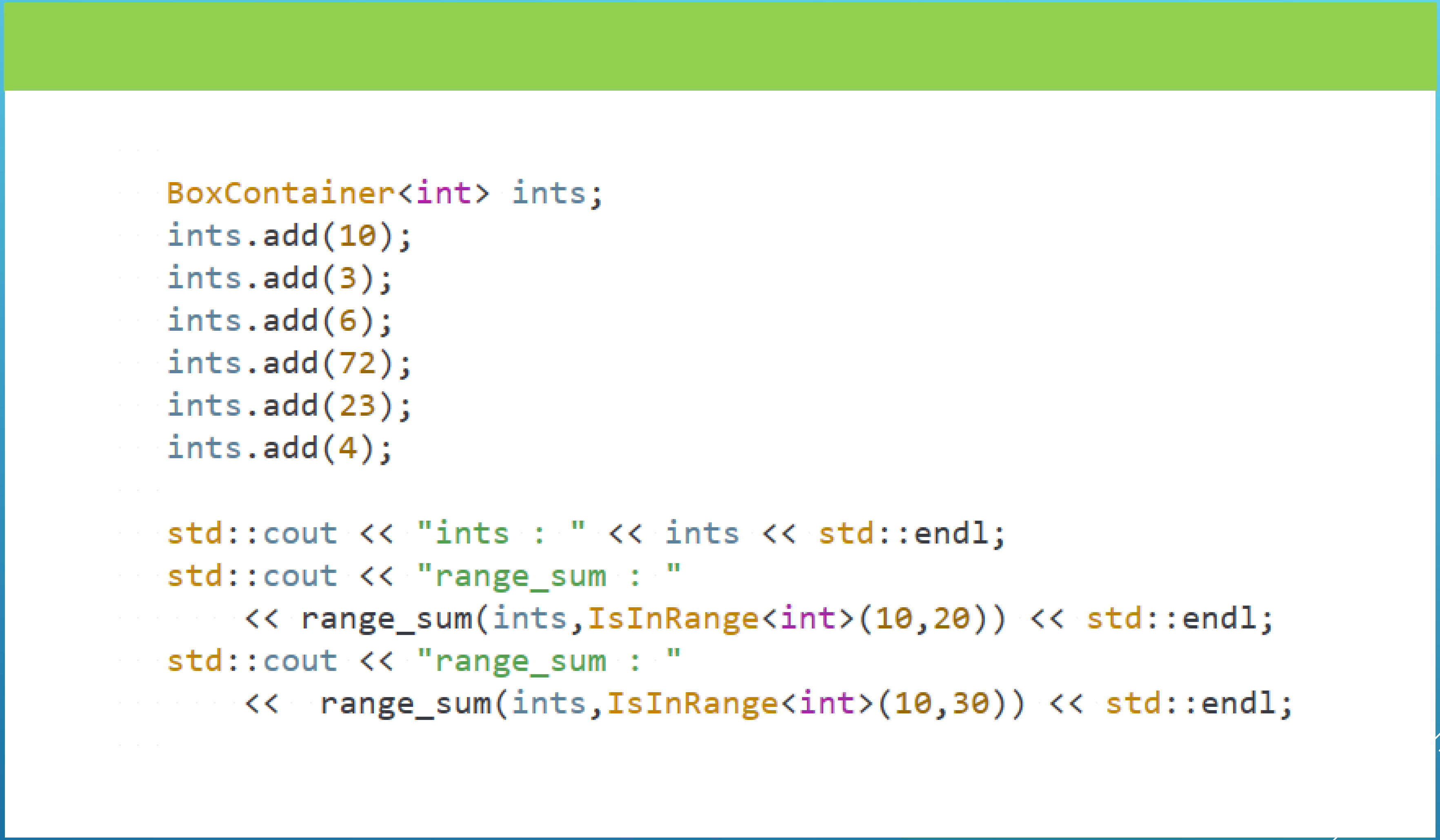
```
//A functor can  take parameters and internally
// store them as member variables
template <typename T>
requires std::is_arithmetic_v<T>
class IsInRange{
public :
    IsInRange(T min, T max) : min_inclusive{min}, max_inclusive{max}{}
    bool operator()(T value) const{
        return ((value >= min_inclusive)&&(value<= max_inclusive));
    }
private :
    T min_inclusive;
    T max_inclusive;
};
```



```cpp
template <typename T ,typename RangePicker>
requires std::is_arithmetic_v<T>
T range_sum (const BoxContainer<T>& collection
                                , RangePicker is_in_range){

    T sum{};
    for(size_t i{}; i < collection.size() ; ++i){
        if(is_in_range(collection.get_item(i)))
            sum += collection.get_item(i);
    }
    return sum;
}
```



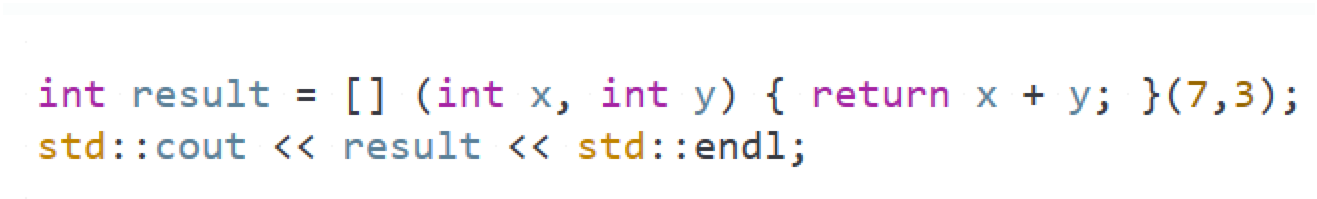
```cpp
BoxContainer<double> doubles;
doubles.add(10.1);
doubles.add(20.2);
doubles.add(30.3);

std::cout << "doubles : " << doubles << std::endl;
std::cout <<"range_sum : "
    <<  range_sum(doubles,IsInRange<double>(10.0,15.5)) << std::endl;
std::cout << "range_sum : "
    <<  range_sum(doubles,IsInRange<double>(10.0,21.5)) << std::endl;

//BoxContainer<std::string> strings;
//strings.add("Hello");
//Compiler error : some constraints not satisfied
//std::cout << range_sum(strings,IsInRange<std::string>("H","W")) << std::endl;
```



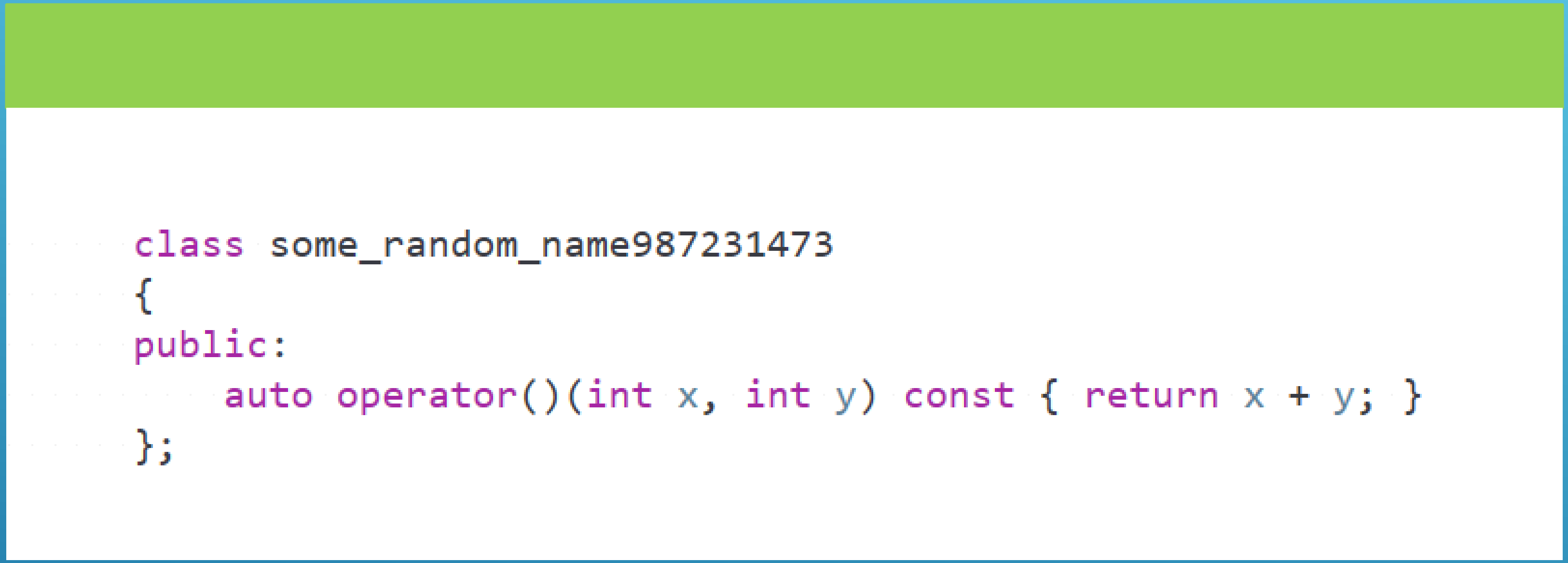
```
BoxContainer<int> ints;
ints.add(10);
ints.add(3);
ints.add(6);
ints.add(72);
ints.add(23);
ints.add(4);

std::cout << "ints : " << ints << std::endl;
std::cout << "range_sum : "
    << range_sum(ints,IsInRange<int>(10,20)) << std::endl;
std::cout << "range_sum : "
    <<  range_sum(ints,IsInRange<int>(10,30)) << std::endl;
```

Slide intentionally left empty

Functors as Lambda functions

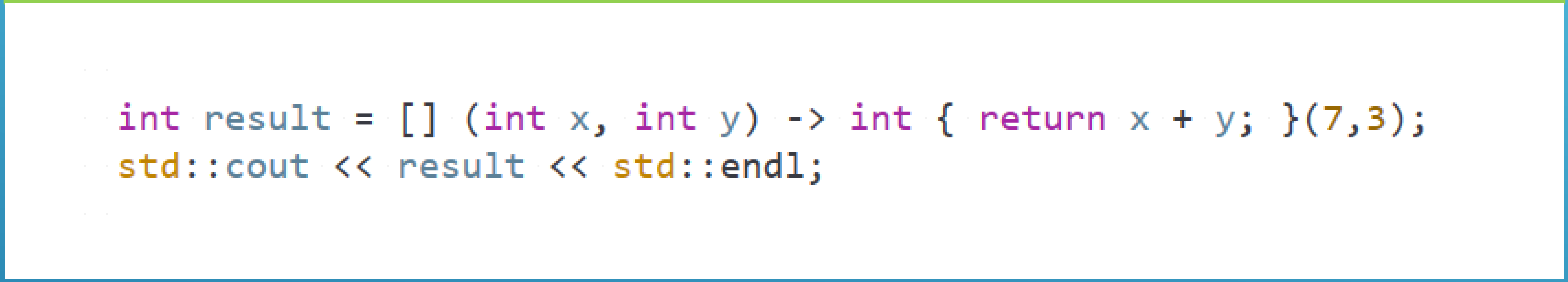


```cpp
int result = [] (int x, int y) { return x + y; }(7,3);
std::cout << result << std::endl;
```



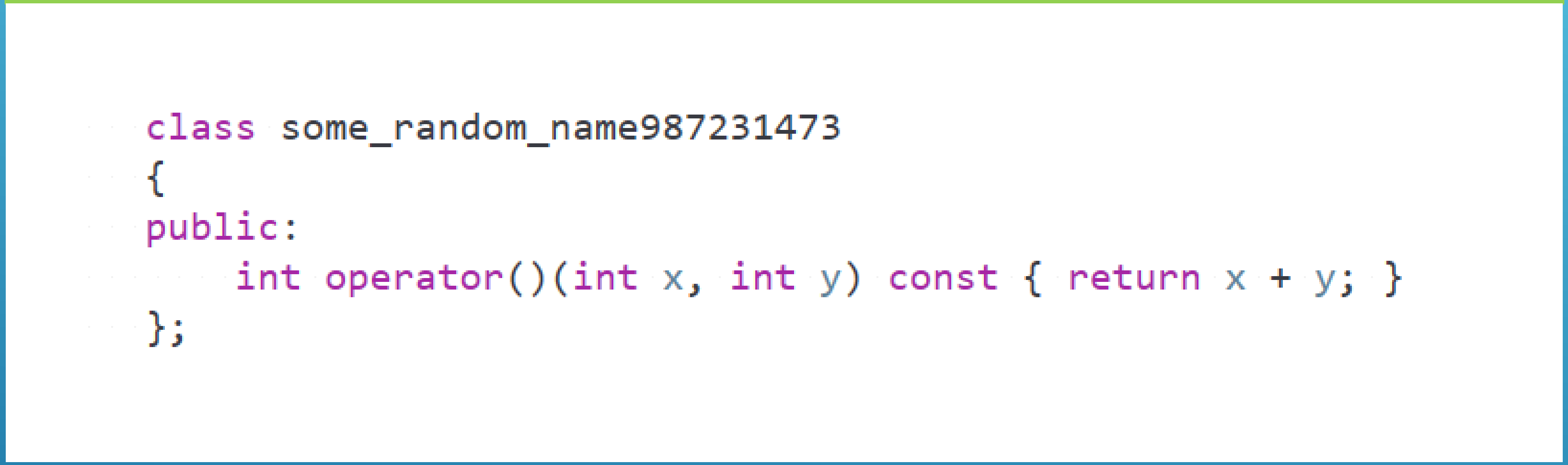
```cpp
class some_random_name987231473
{
public:
    auto operator()(int x, int y) const { return x + y; }
};
```

Specifying the return type



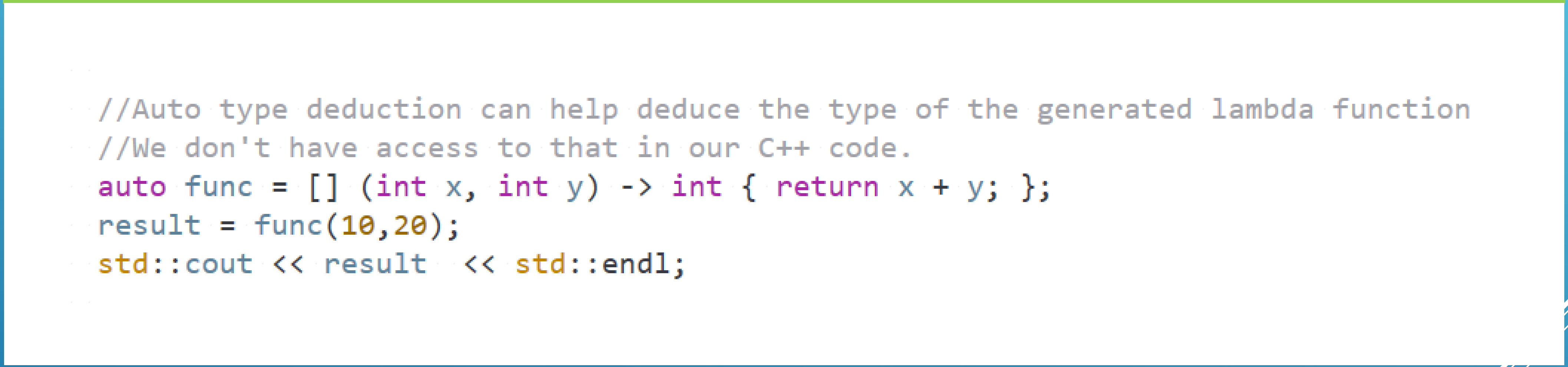
```
int result = [] (int x, int y) -> int { return x + y; }(7,3);
std::cout << result << std::endl;
```

Specifying the return type



```
class some_random_name987231473
{
public:
    int operator()(int x, int y) const { return x + y; }
};
```

Auto deducing the type of the lambda function generated by the compiler

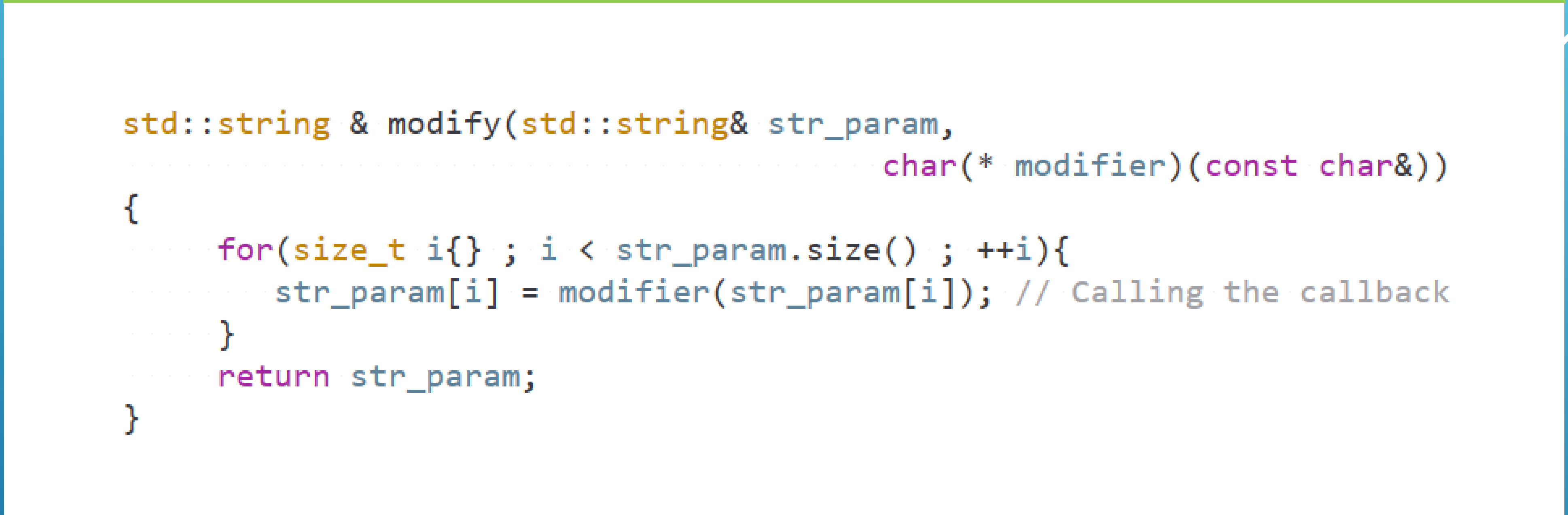


```
//Auto type deduction can help deduce the type of the generated lambda function
//We don't have access to that in our C++ code.
auto func = [] (int x, int y) -> int { return x + y; };
result = func(10,20);
std::cout << result  << std::endl;
```

Slide intentionally left empty

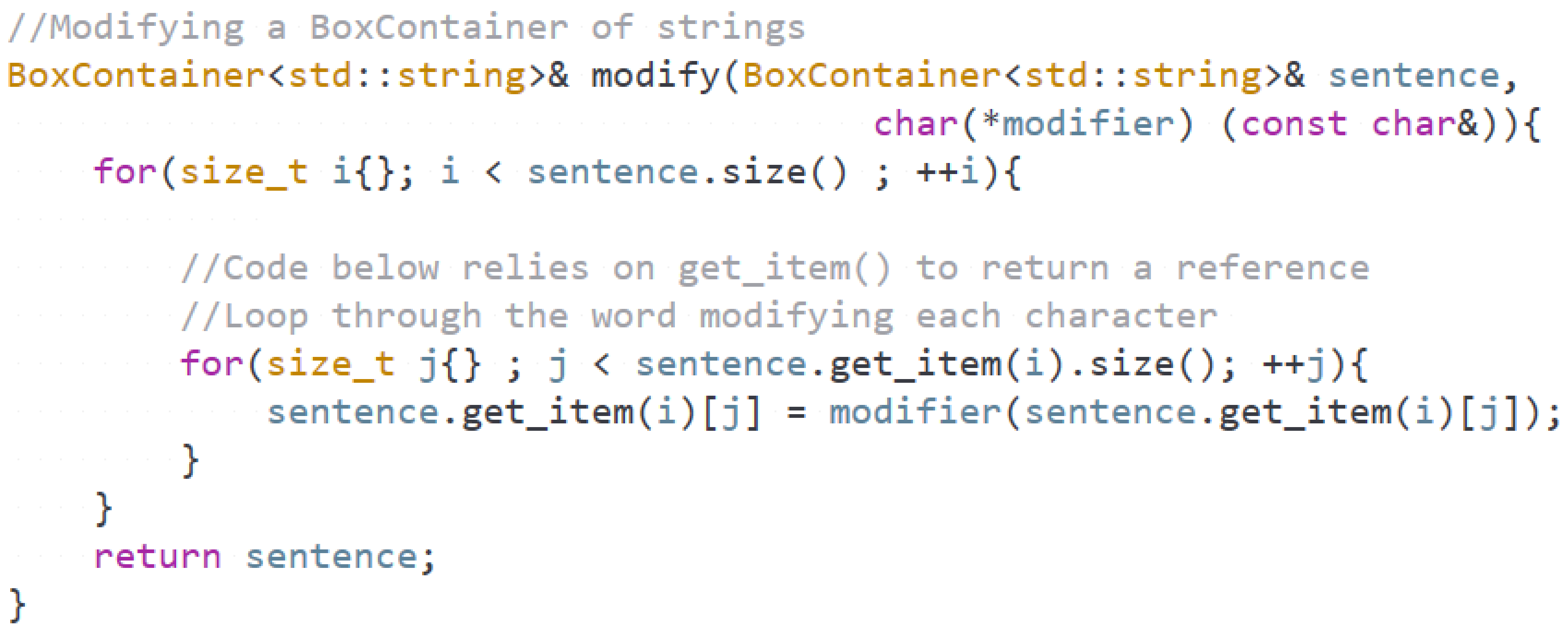
Lambda functions as callbacks

Modifying a std::string



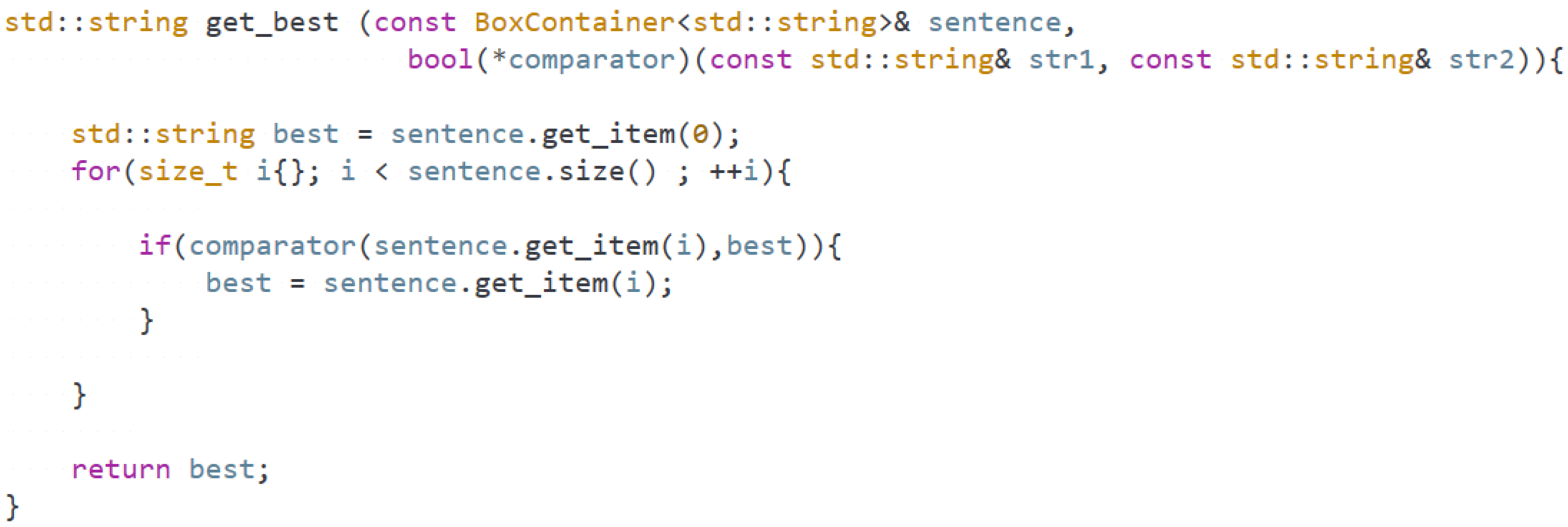
```
std::string & modify(std::string& str_param,
                                         char(* modifier)(const char&))
{
     for(size_t i{} ; i < str_param.size() ; ++i){
        str_param[i] = modifier(str_param[i]); // Calling the callback
     }
     return str_param;
}
```

Modifying a BoxContainer of std::string's



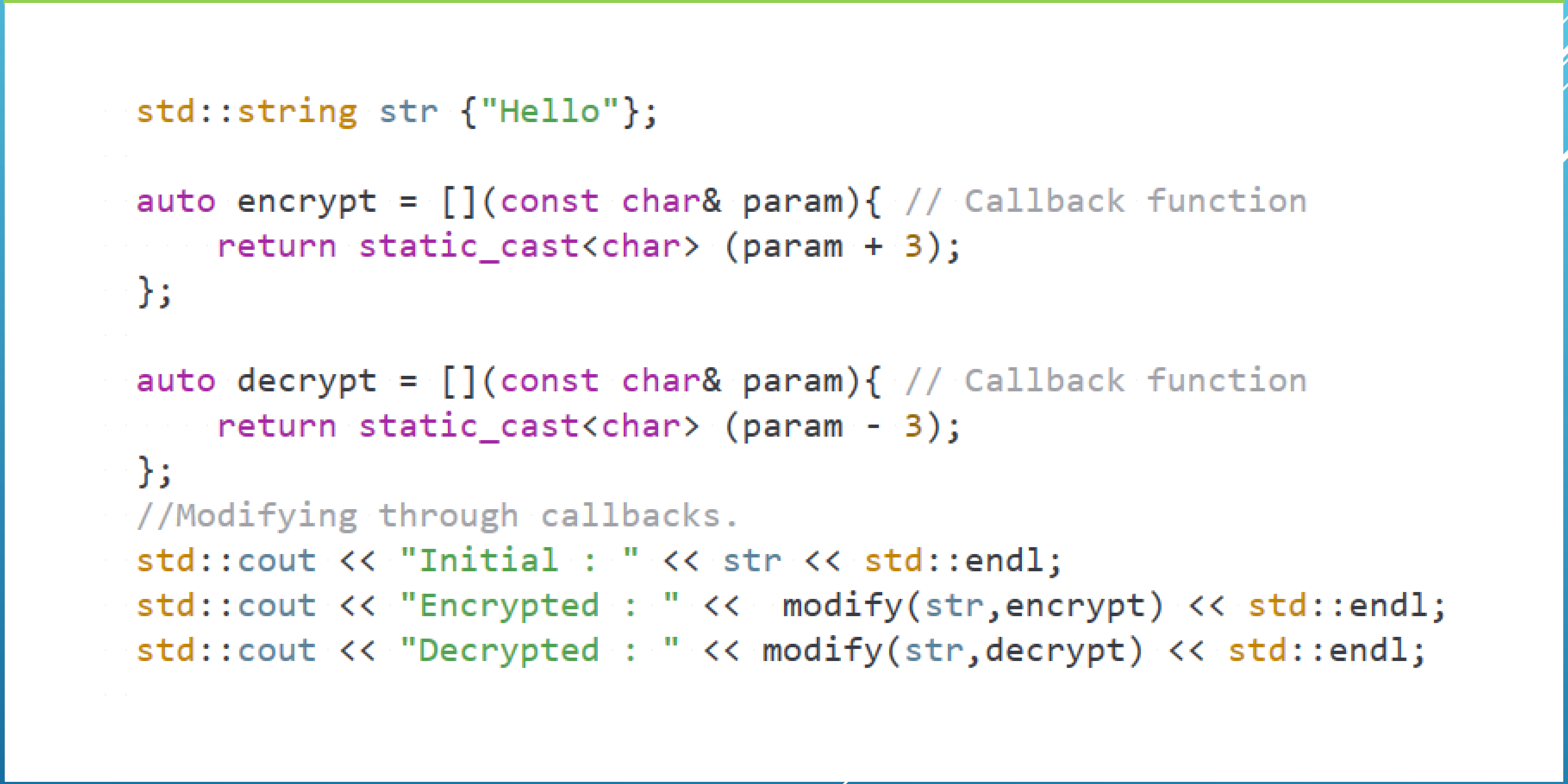
```
//Modifying a BoxContainer of strings
BoxContainer<std::string>& modify(BoxContainer<std::string>& sentence,
                                        char(*modifier) (const char&)){
    for(size_t i{}; i < sentence.size() ; ++i){

        //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;
}
```

Getting the best



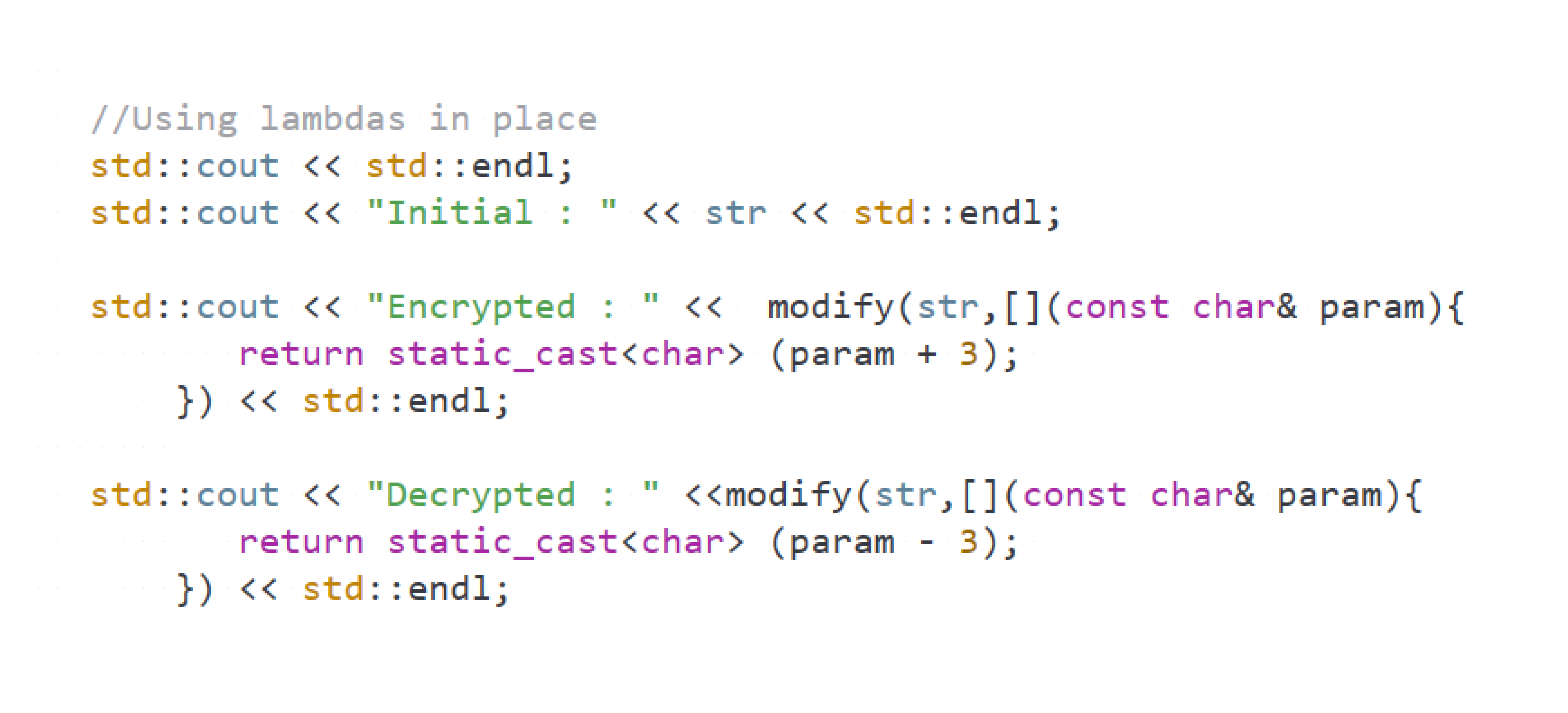
```cpp
std::string get_best (const BoxContainer<std::string>& sentence,
                        bool(*comparator)(const std::string& str1, const std::string& str2)){

    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;
}
```

Lambda function callbacks



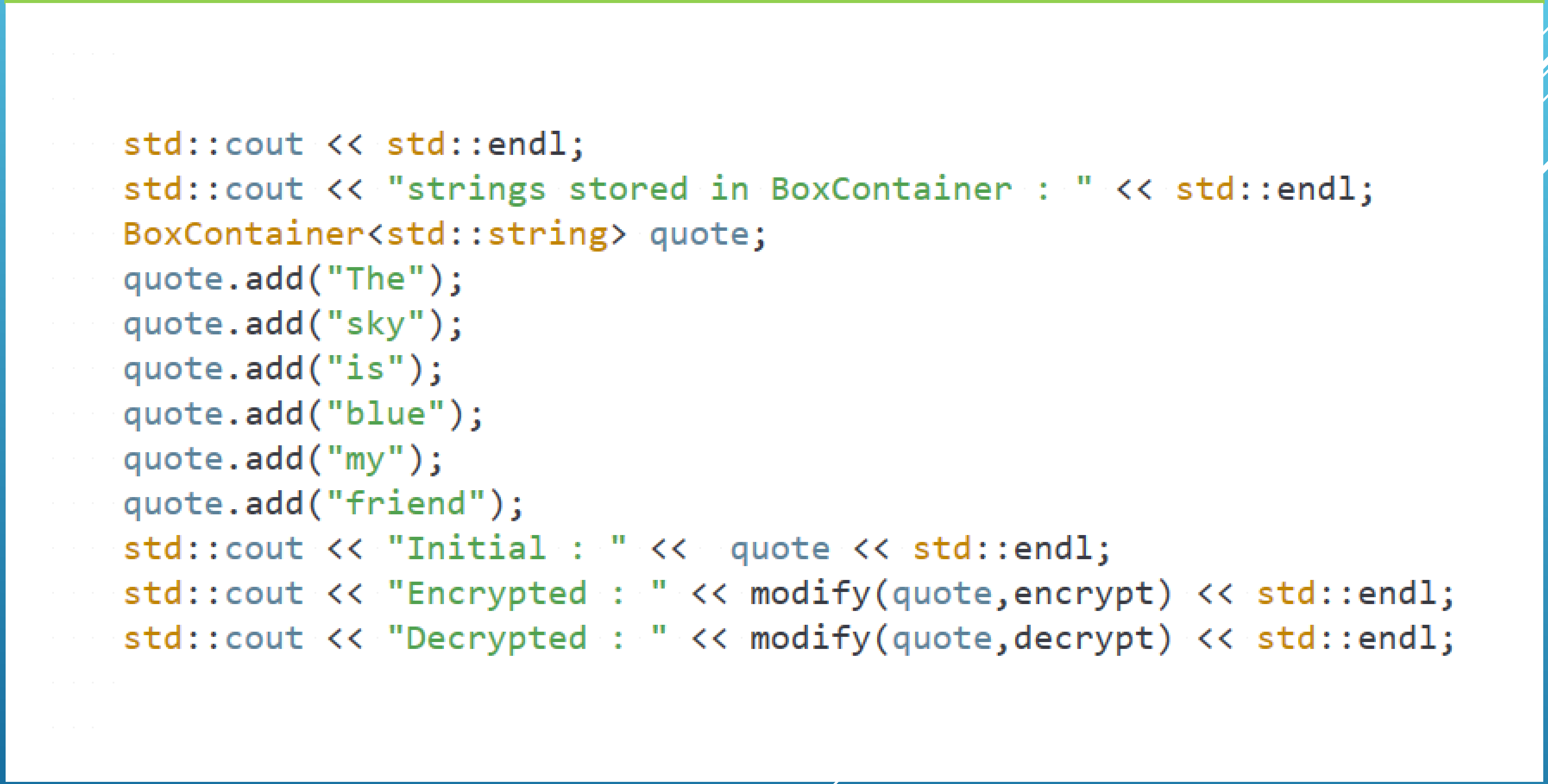
```cpp
std::string str {"Hello"};

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.
std::cout << "Initial : " << str << std::endl;
std::cout << "Encrypted : " <<  modify(str,encrypt) << std::endl;
std::cout << "Decrypted : " << modify(str,decrypt) << std::endl;
```

Anonymous lambda function callbacks



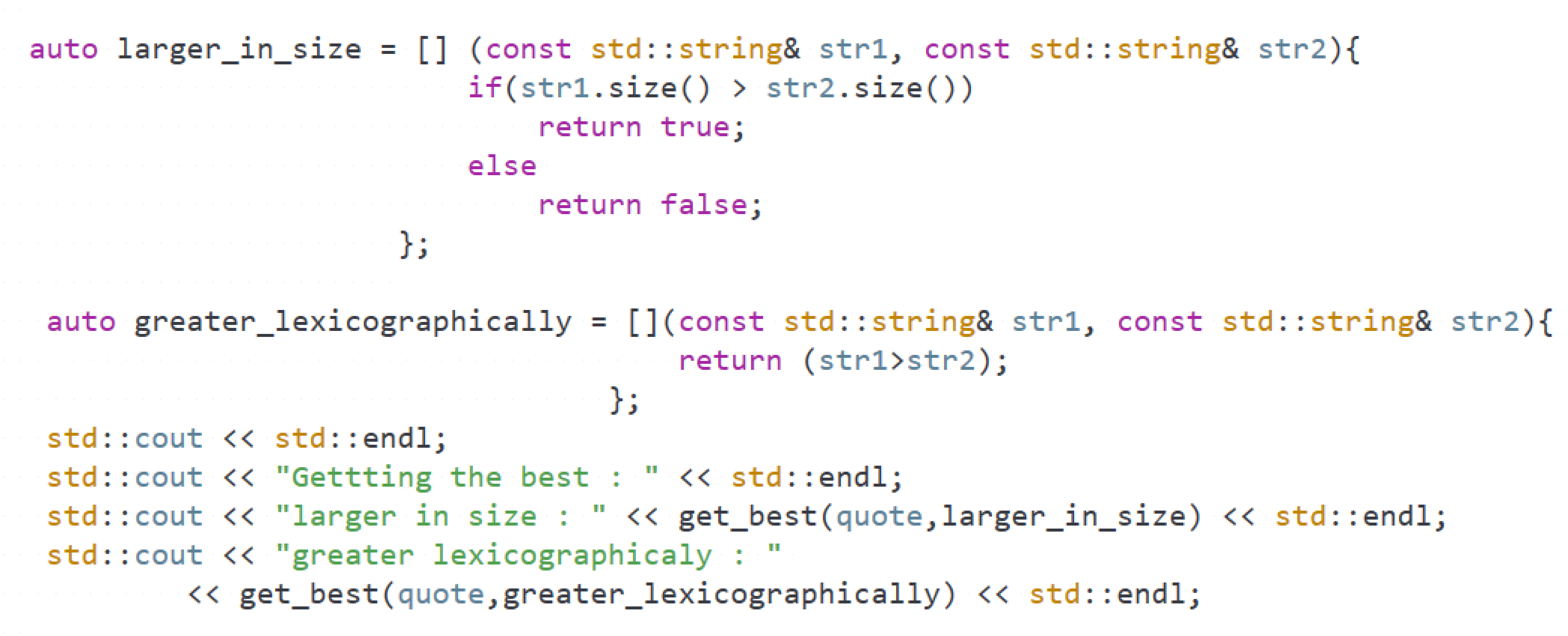
```
//Using lambdas in place
std::cout << std::endl;
std::cout << "Initial : " << str << std::endl;

std::cout << "Encrypted : " <<  modify(str,[](const char& param){
       return static_cast<char> (param + 3);
    }) << std::endl;

std::cout << "Decrypted : " <<modify(str,[](const char& param){
       return static_cast<char> (param - 3);
    }) << std::endl;
```

std::string’s in a BoxContainer



```
std::cout << std::endl;
std::cout << "strings stored in BoxContainer : " << std::endl;
BoxContainer<std::string> quote;
quote.add("The");
quote.add("sky");
quote.add("is");
quote.add("blue");
quote.add("my");
quote.add("friend");
std::cout << "Initial : " <<  quote << std::endl;
std::cout << "Encrypted : " << modify(quote,encrypt) << std::endl;
std::cout << "Decrypted : " << modify(quote,decrypt) << std::endl;
```

Getting the best

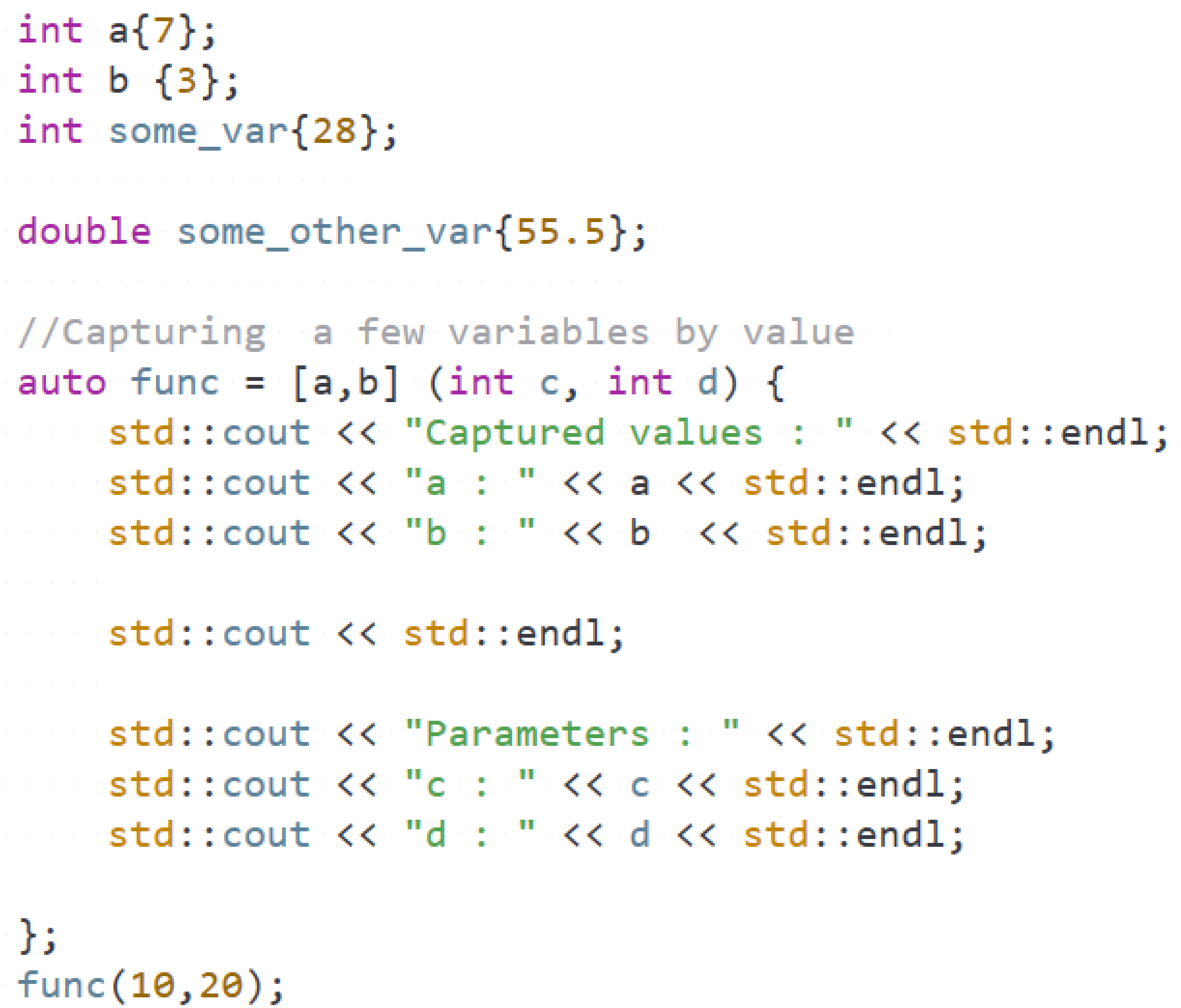


```
auto larger_in_size = [] (const std::string& str1, const std::string& str2){
                         if(str1.size() > str2.size())
                             return true;
                         else
                             return false;
                     };

 auto greater_lexicographically = [](const std::string& str1, const std::string& str2){
                                     return (str1>str2);
                                 };
 std::cout << std::endl;
 std::cout << "Gettting the best : " << std::endl;
 std::cout << "larger in size : " << get_best(quote,larger_in_size) << std::endl;
 std::cout << "greater lexicographicaly : "
         << get_best(quote,greater_lexicographically) << std::endl;
```

Slide intentionally left empty

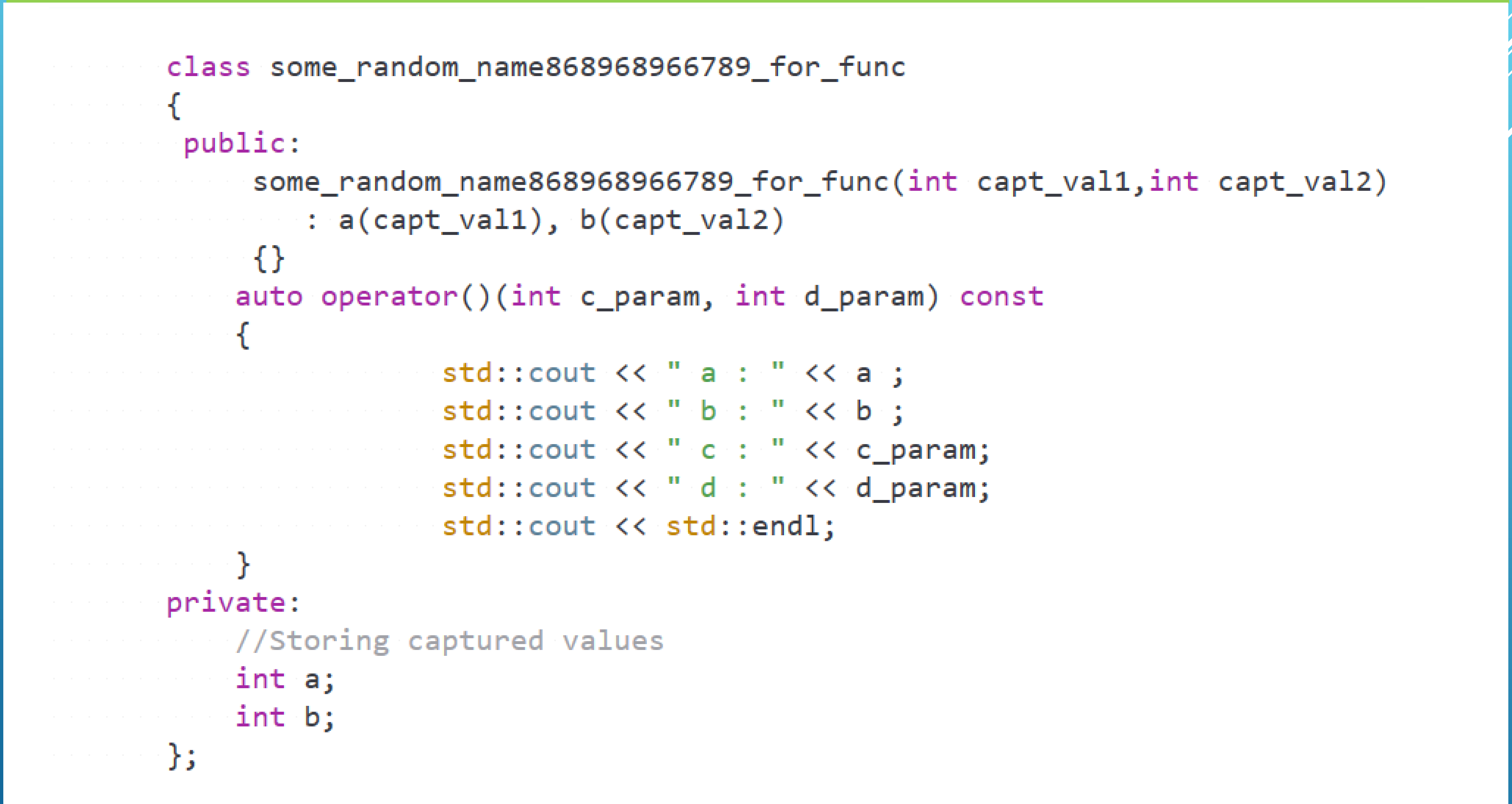
Capturing by value under the hood

Capturing some variables by value



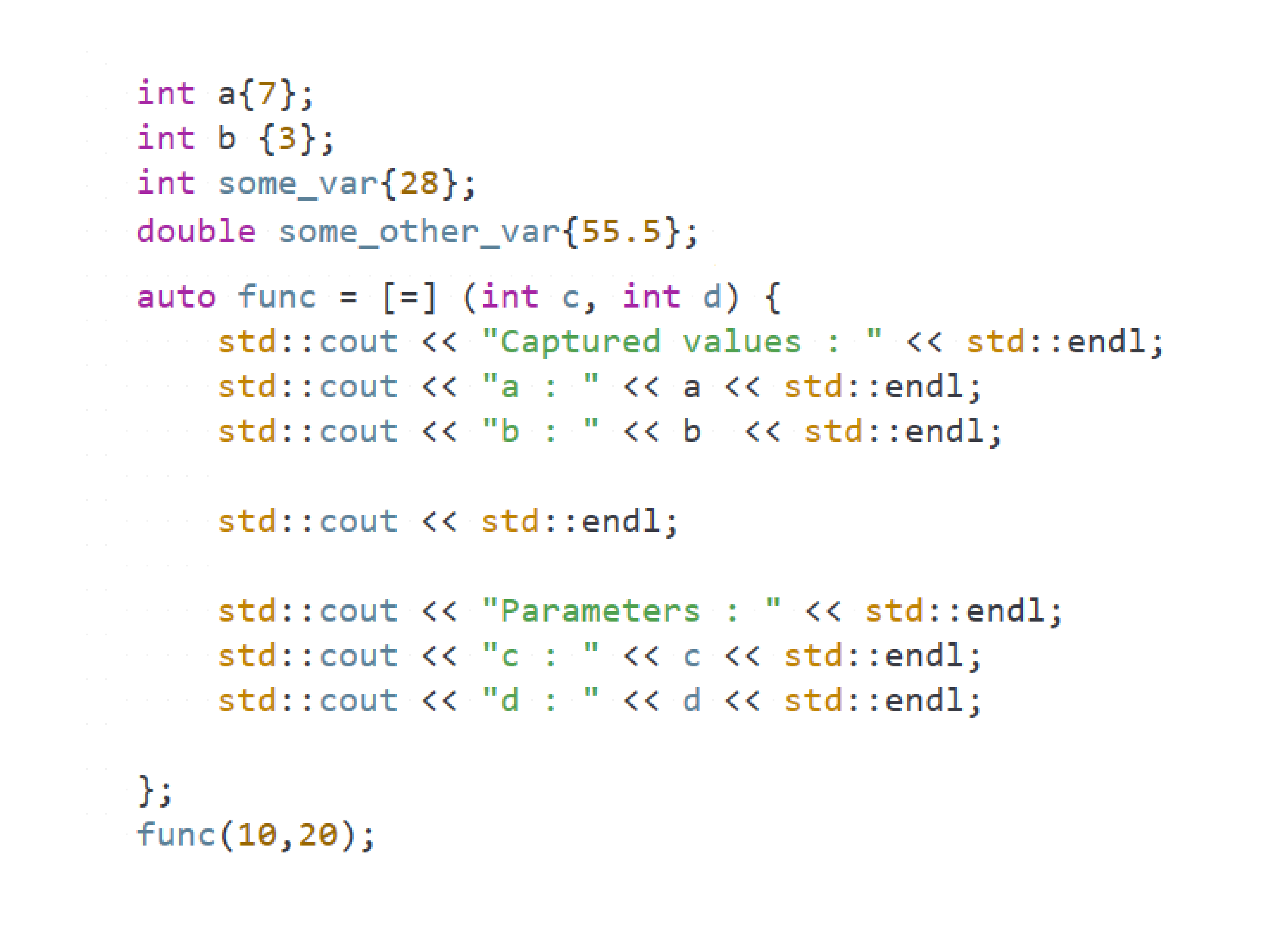
```
int a{7};
int b {3};
int some_var{28};

double some_other_var{55.5};

//Capturing  a few variables by value
auto func = [a,b] (int c, int d) {
    std::cout << "Captured values : " << std::endl;
    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b  << std::endl;

    std::cout << std::endl;

    std::cout << "Parameters : " << std::endl;
    std::cout << "c : " << c << std::endl;
    std::cout << "d : " << d << std::endl;

};
func(10,20);
```

Compiler generated functor



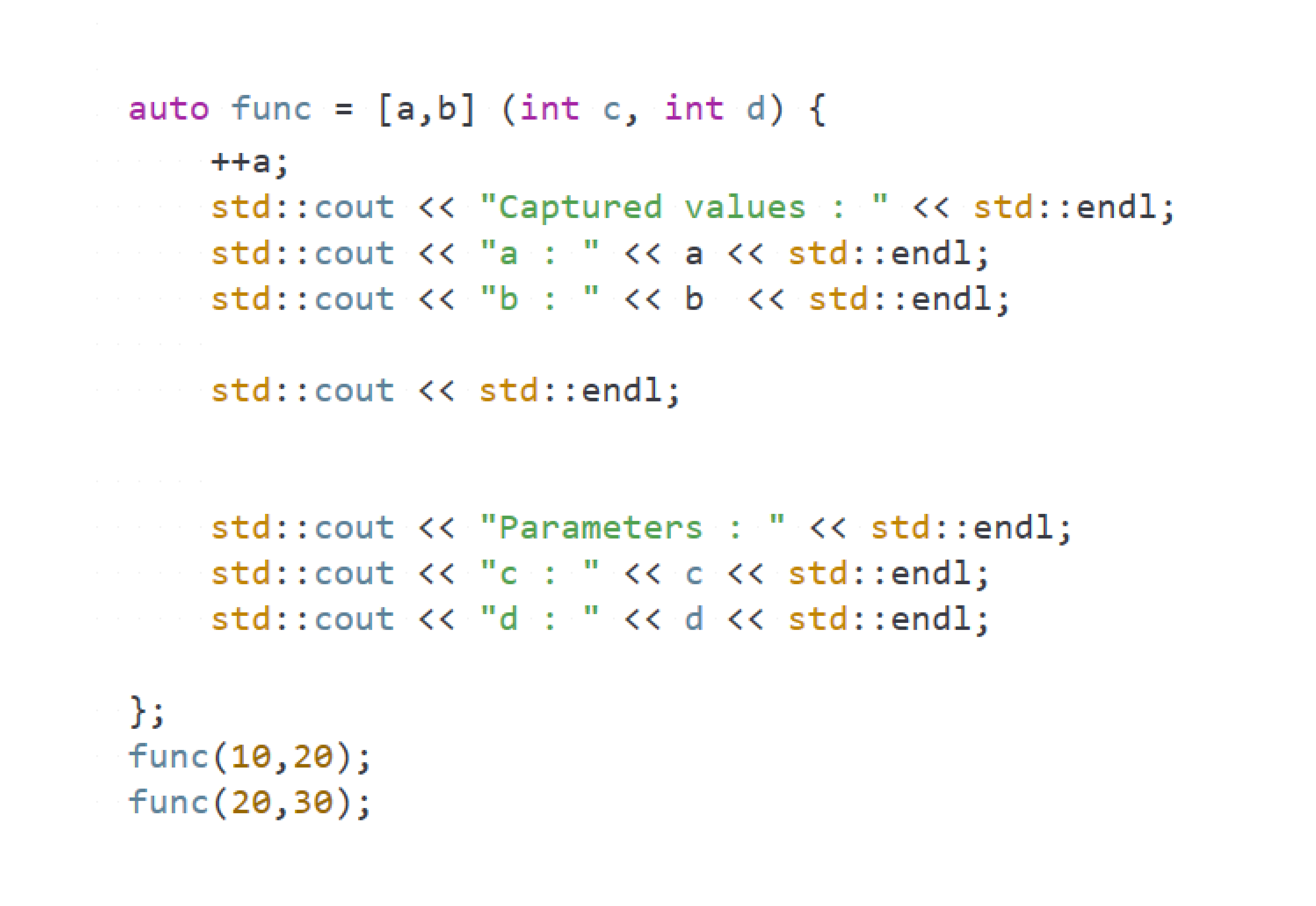
```cpp
class some_random_name868968966789_for_func
{
 public:
   some_random_name868968966789_for_func(int capt_val1,int capt_val2)
       : a(capt_val1), b(capt_val2)
   {}
  auto operator()(int c_param, int d_param) const
  {
        std::cout << " a : " << a ;
        std::cout << " b : " << b ;
        std::cout << " c : " << c_param;
        std::cout << " d : " << d_param;
        std::cout << std::endl;
  }
private:
  //Storing captured values
  int a;
  int b;
};
```

Capturing all variables by value



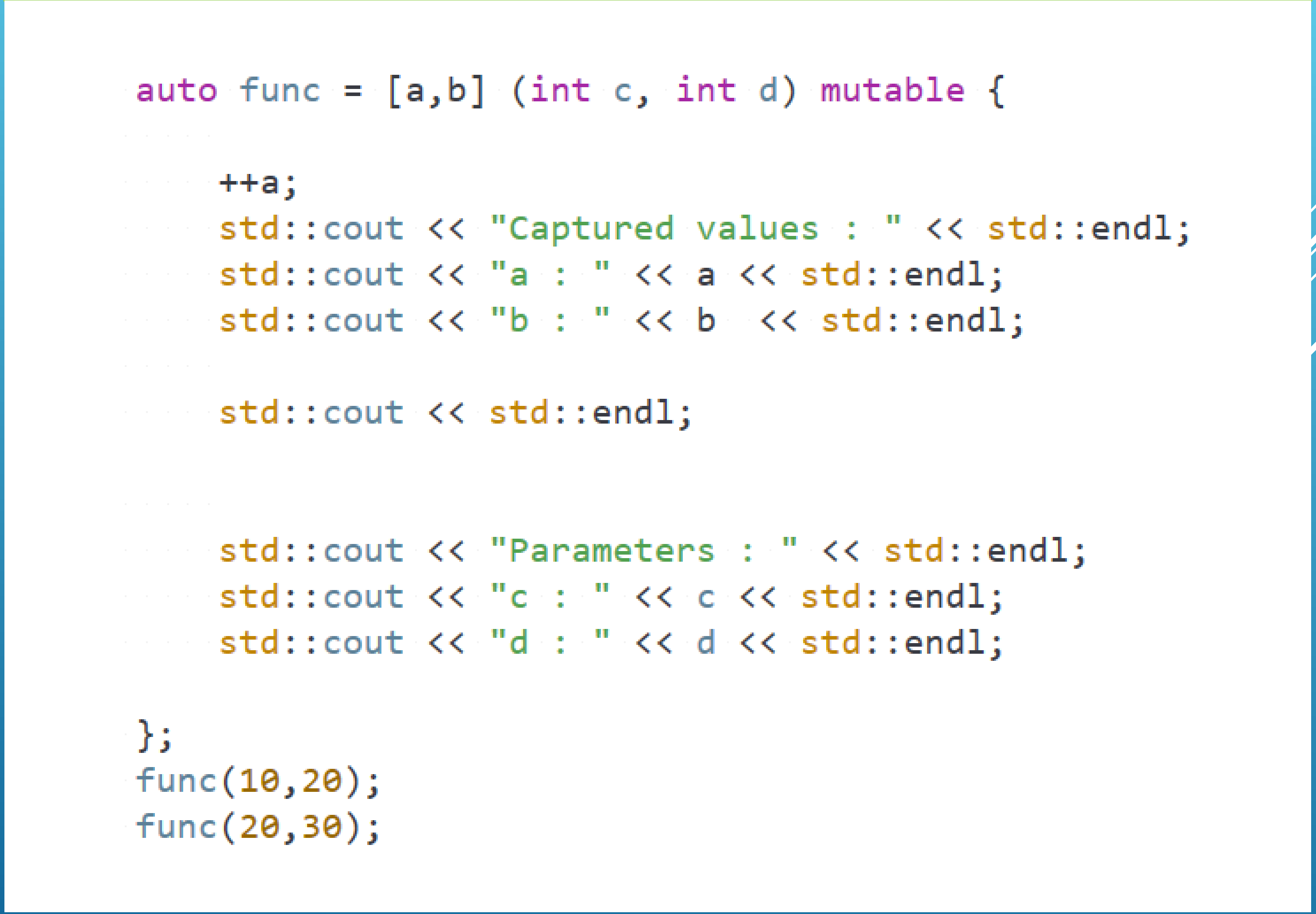
```cpp
int a{7};
int b {3};
int some_var{28};
double some_other_var{55.5};

auto func = [=] (int c, int d) {
    std::cout << "Captured values : " << std::endl;
    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b  << std::endl;

    std::cout << std::endl;

    std::cout << "Parameters : " << std::endl;
    std::cout << "c : " << c << std::endl;
    std::cout << "d : " << d << std::endl;

};
func(10,20);
```

Modifying captured data



```cpp
auto func = [a,b] (int c, int d) {
    ++a;
    std::cout << "Captured values : " << std::endl;
    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b  << std::endl;

    std::cout << std::endl;


    std::cout << "Parameters : " << std::endl;
    std::cout << "c : " << c << std::endl;
    std::cout << "d : " << d << std::endl;

};
func(10,20);
func(20,30);
```

Modifying captured data

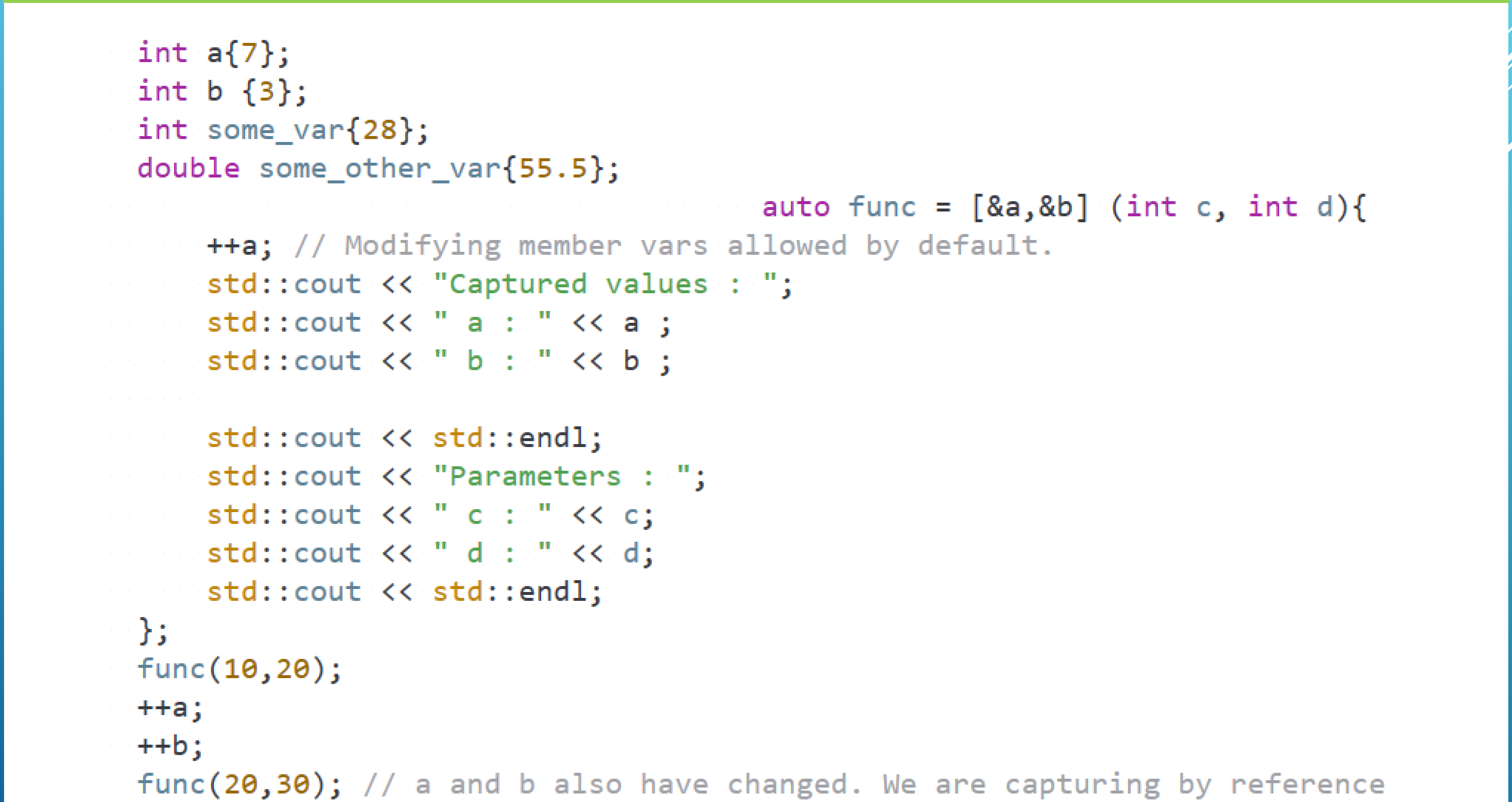


```cpp
auto func = [a,b] (int c, int d) mutable {

    ++a;
    std::cout << "Captured values : " << std::endl;
    std::cout << "a : " << a << std::endl;
    std::cout << "b : " << b  << std::endl;

    std::cout << std::endl;


    std::cout << "Parameters : " << std::endl;
    std::cout << "c : " << c << std::endl;
    std::cout << "d : " << d << std::endl;

};
func(10,20);
func(20,30);
```

Slide intentionally left empty

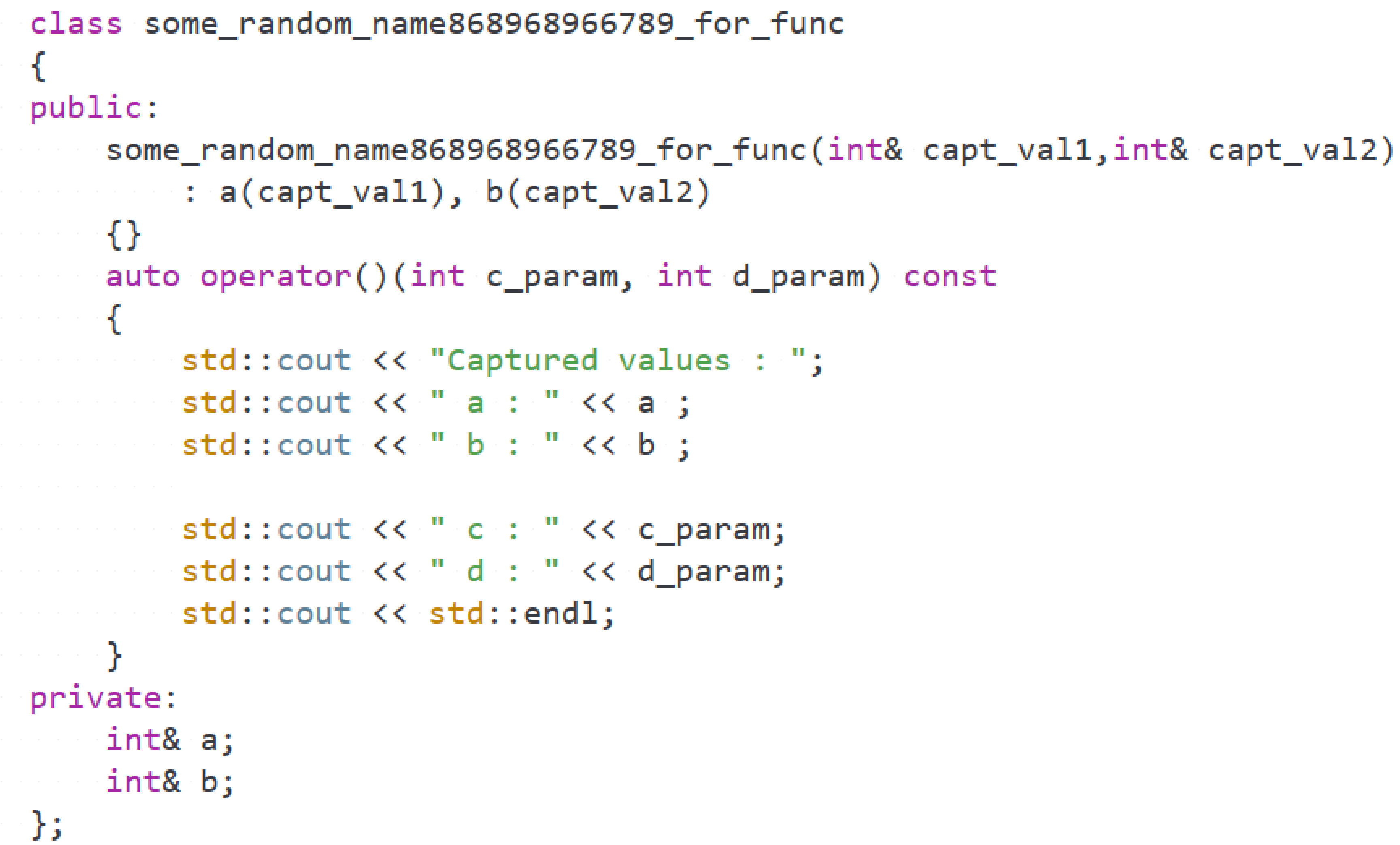
Capturing by reference under the hood

Capturing some variables by reference



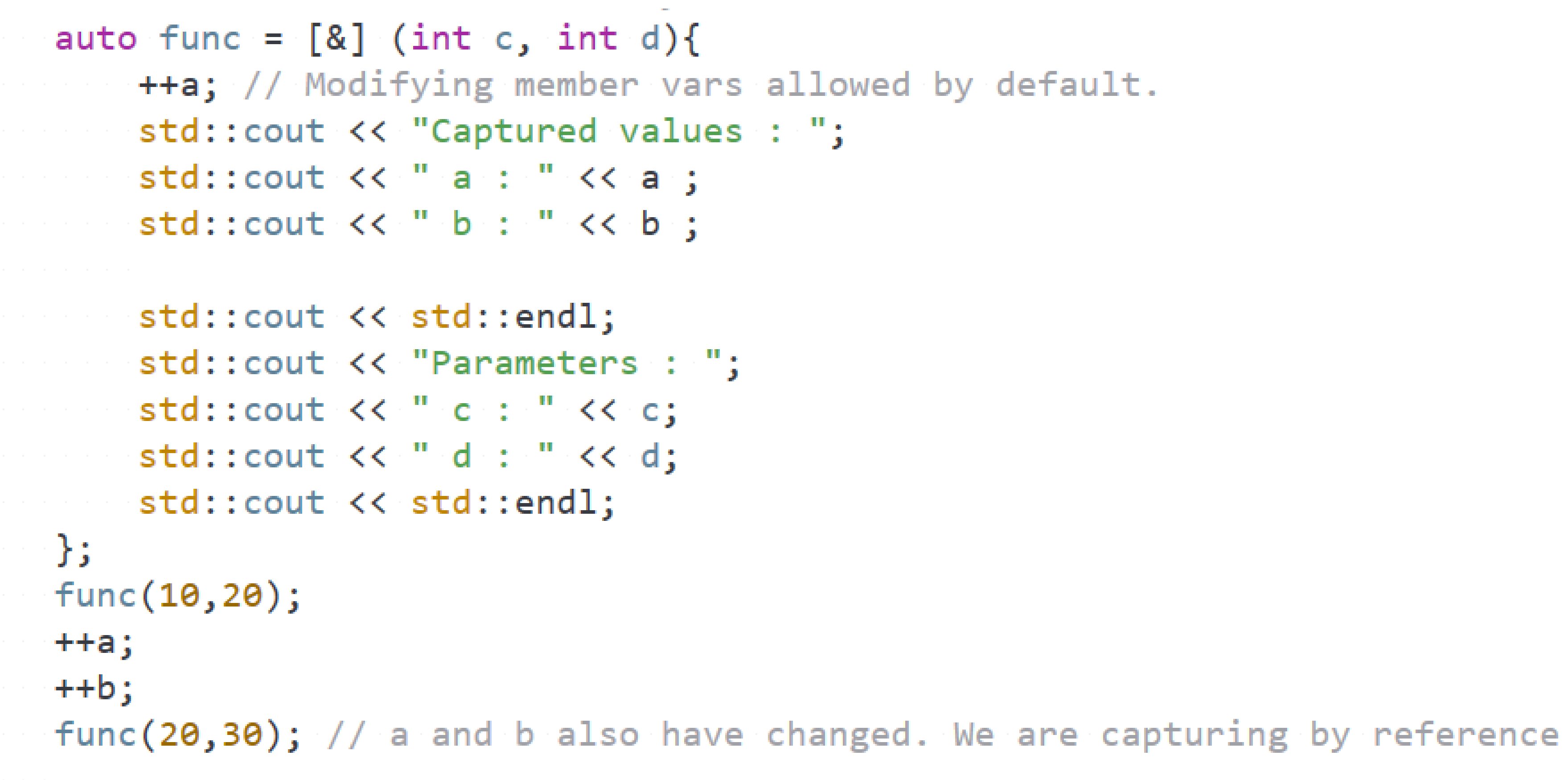
```
int a{7};
int b {3};
int some_var{28};
double some_other_var{55.5};
                                    auto func = [&a,&b] (int c, int d){
    ++a; // Modifying member vars allowed by default.
    std::cout << "Captured values : ";
    std::cout << " a : " << a ;
    std::cout << " b : " << b ;

    std::cout << std::endl;
    std::cout << "Parameters : ";
    std::cout << " c : " << c;
    std::cout << " d : " << d;
    std::cout << std::endl;
};
func(10,20);
++a;
++b;
func(20,30); // a and b also have changed. We are capturing by reference
```

Compiler generated functor



```cpp
class some_random_name868968966789_for_func
{
public:
    some_random_name868968966789_for_func(int& capt_val1,int& capt_val2)
        : a(capt_val1), b(capt_val2)
    {}
    auto operator()(int c_param, int d_param) const
    {
        std::cout << "Captured values : ";
        std::cout << " a : " << a ;
        std::cout << " b : " << b ;

        std::cout << " c : " << c_param;
        std::cout << " d : " << d_param;
        std::cout << std::endl;
    }
private:
    int& a;
    int& b;
};
```

Capturing all variables by reference



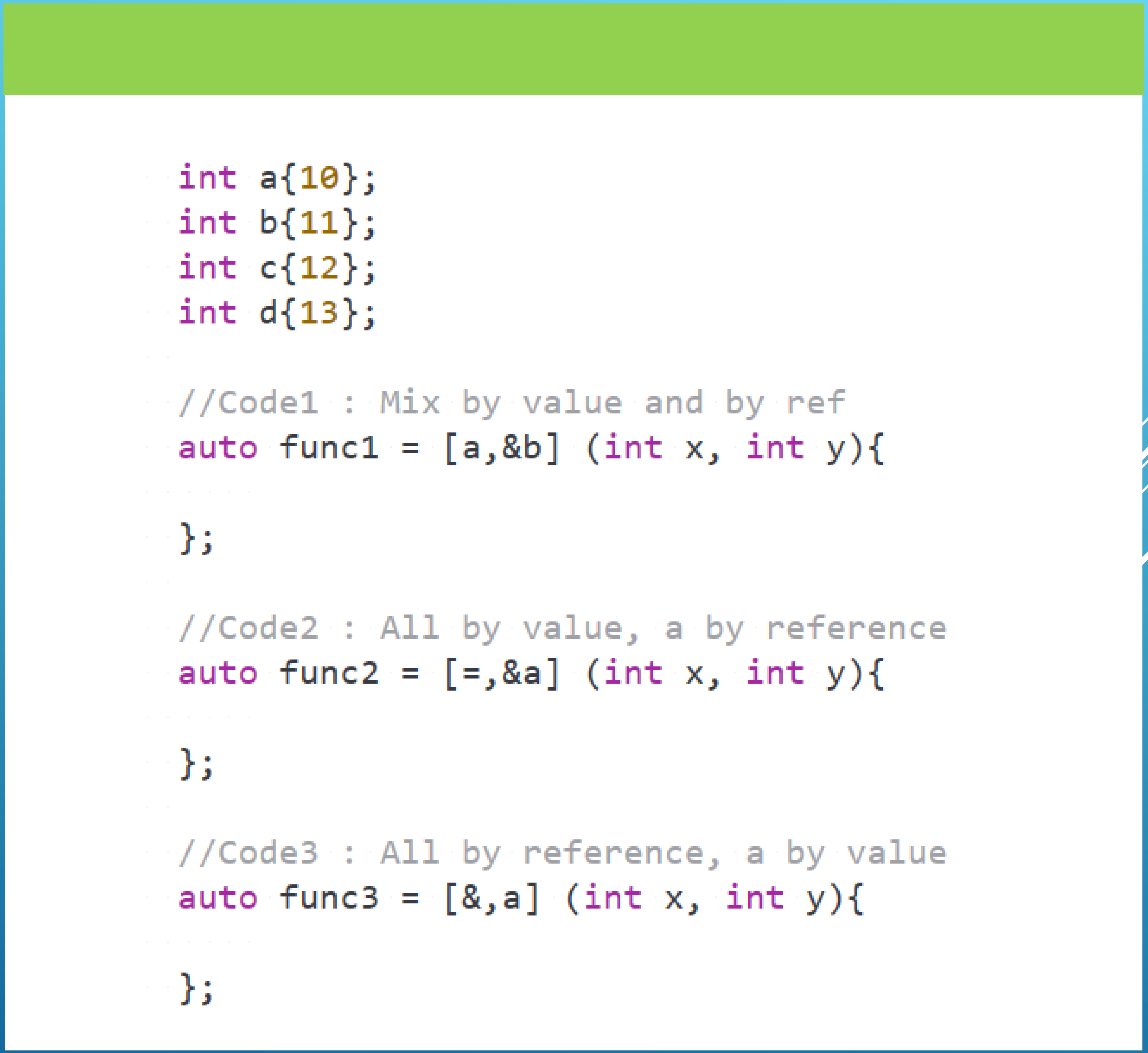
```
auto func = [&] (int c, int d){
    ++a; // Modifying member vars allowed by default.
    std::cout << "Captured values : ";
    std::cout << " a : " << a ;
    std::cout << " b : " << b ;

    std::cout << std::endl;
    std::cout << "Parameters : ";
    std::cout << " c : " << c;
    std::cout << " d : " << d;
    std::cout << std::endl;
};
func(10,20);
++a;
++b;
func(20,30); // a and b also have changed. We are capturing by reference
```

Modifying captured data

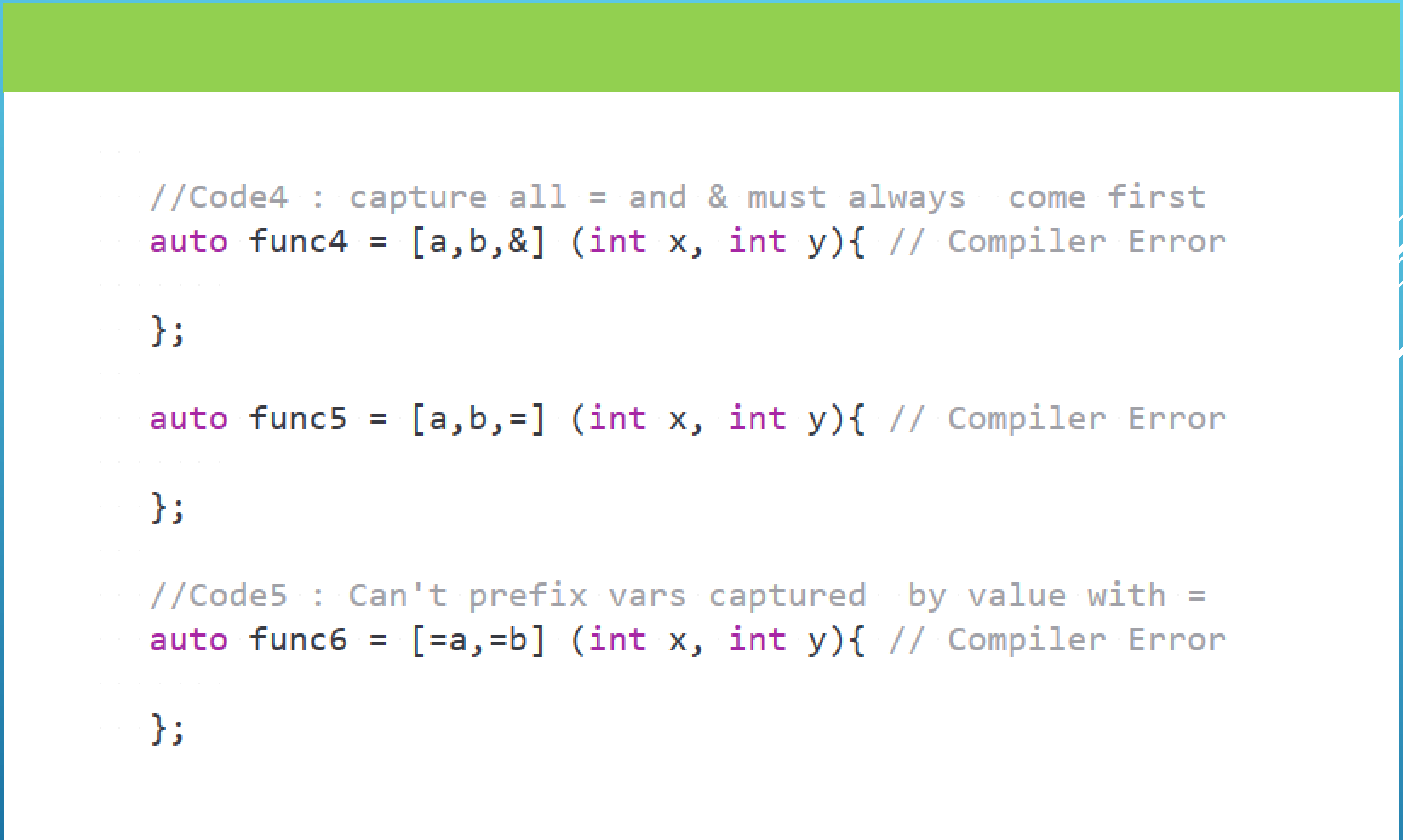
Modifying member variables from the lambda function body is allowed

Slide intentionally left empty

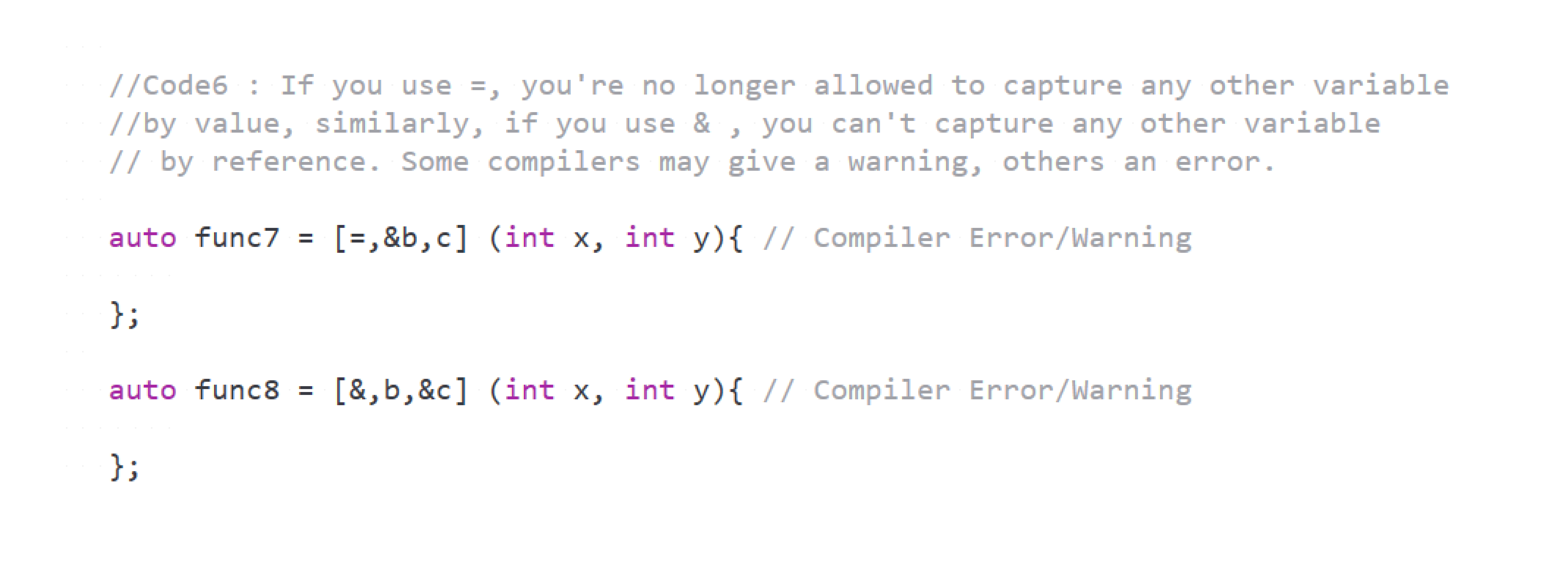
Mixin capturing



```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){

};
```



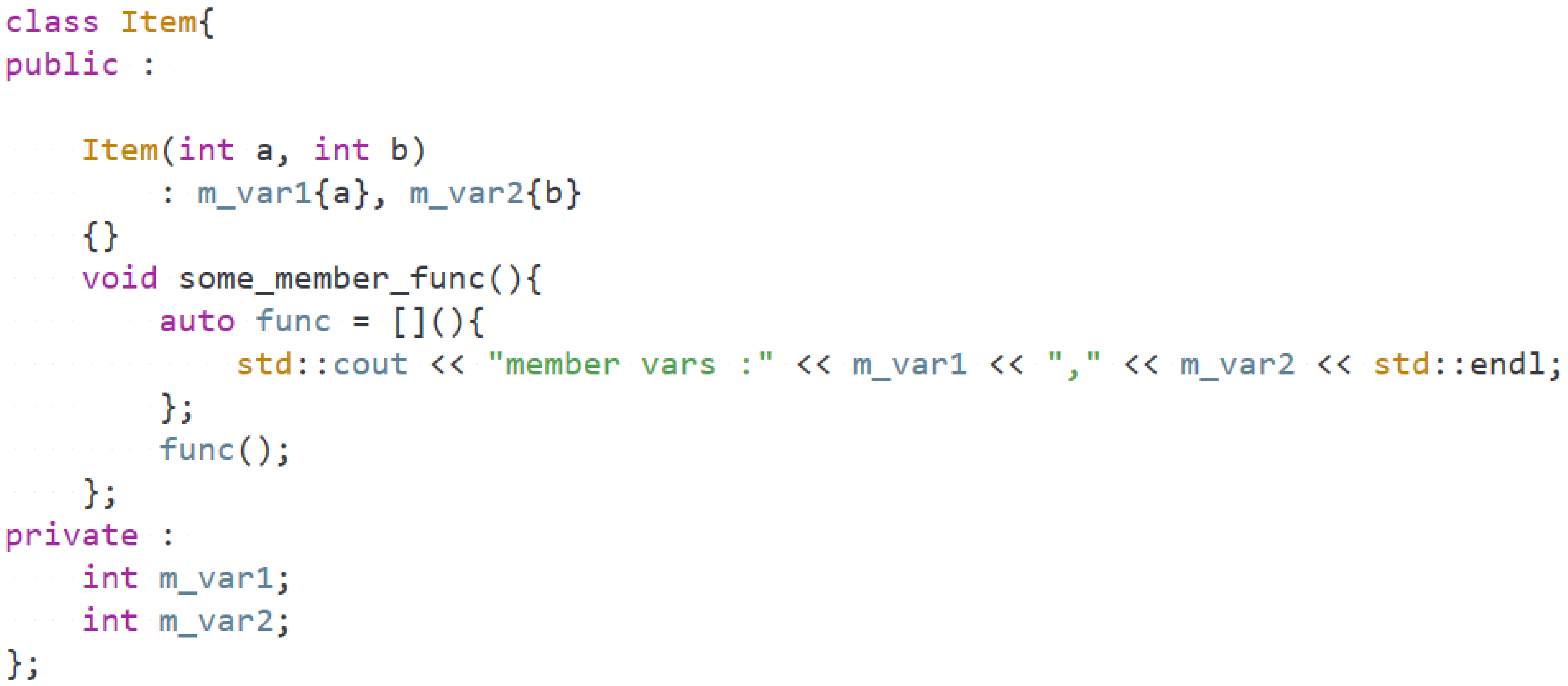
```
//Code4 : capture all = and & must always  come first
auto func4 = [a,b,&] (int x, int y){ // Compiler Error

};

auto func5 = [a,b,=] (int x, int y){ // Compiler Error

};

//Code5 : Can't prefix vars captured  by value with =
auto func6 = [=a,=b] (int x, int y){ // Compiler Error

};
```



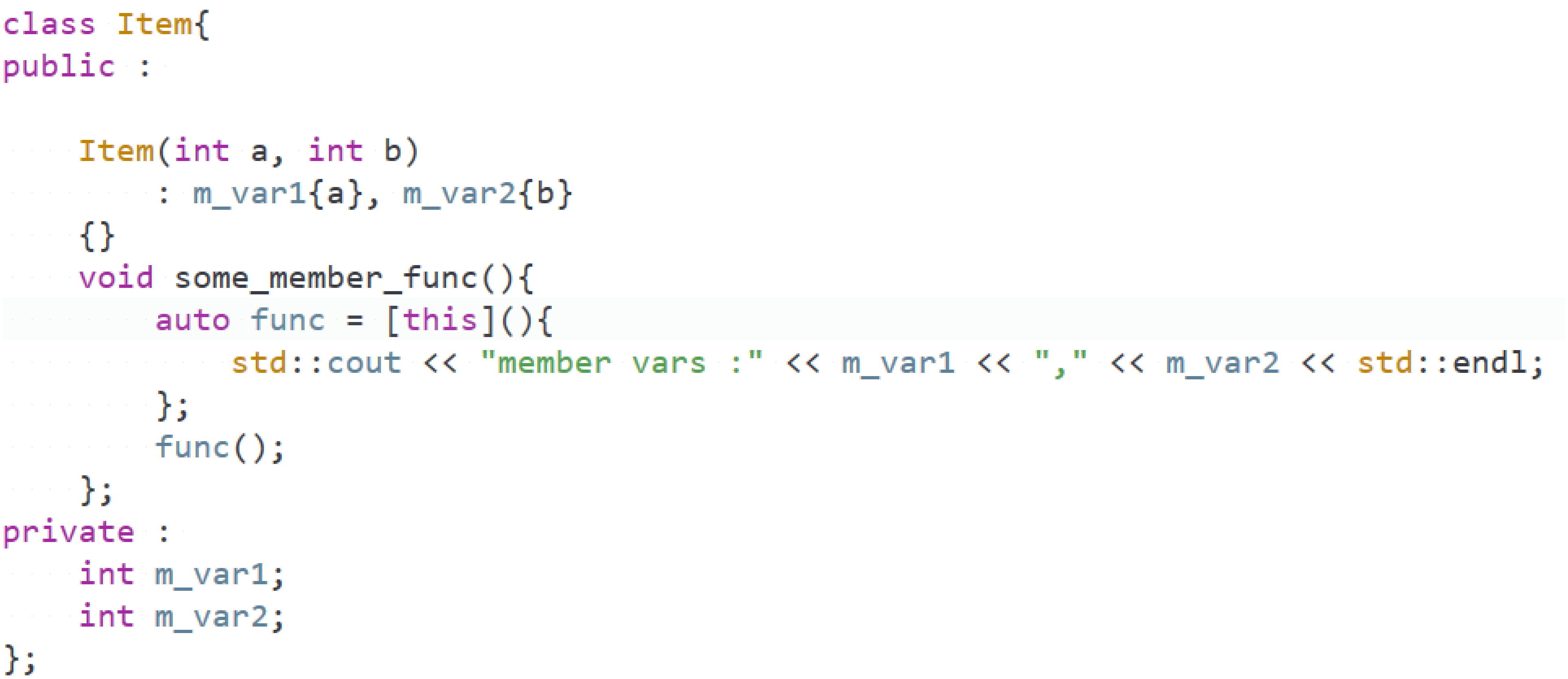
```cpp
//Code6 : If you use =, you're no longer allowed to capture any other variable
//by value, similarly, if you use & , you can't capture any other variable
// by reference. Some compilers may give a warning, others an error.

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

};

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

};
```

Slide intentionally left empty

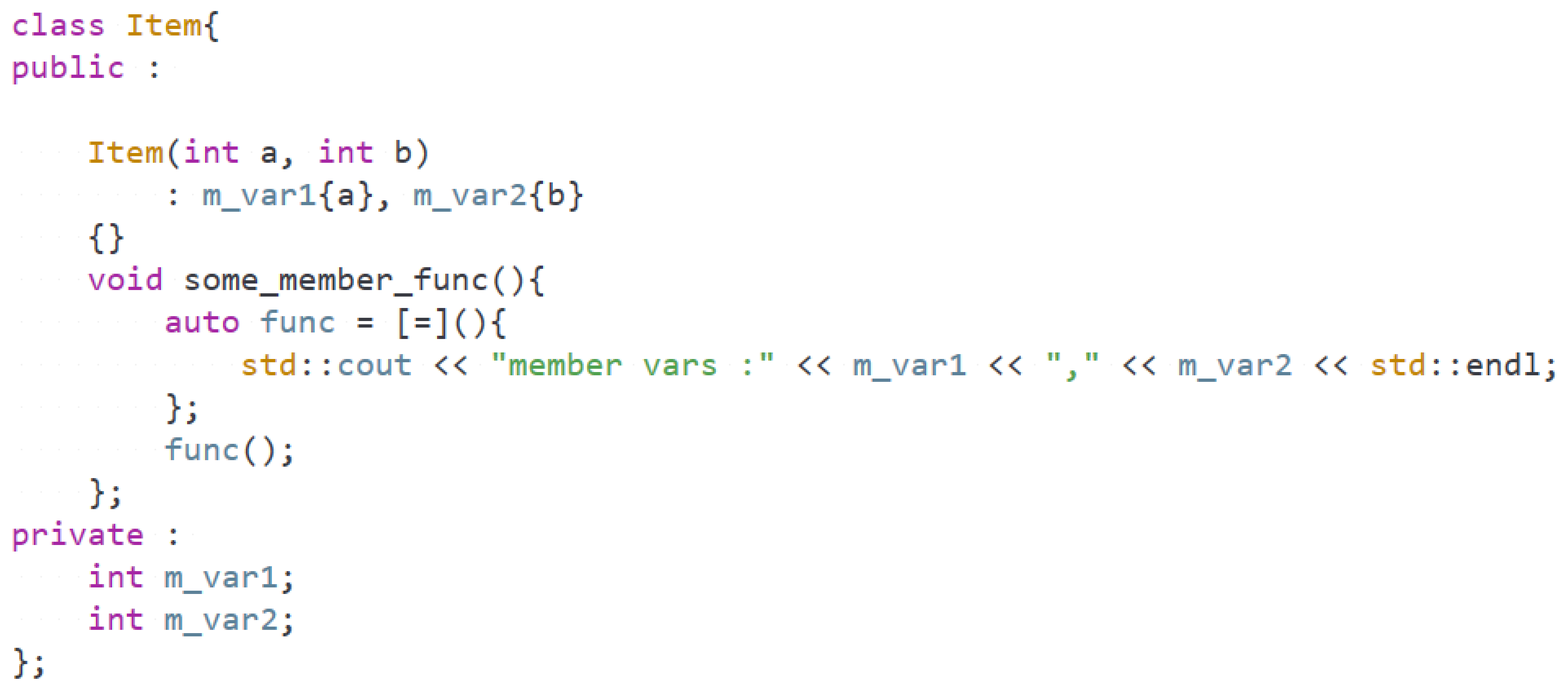
Capturing the this pointer



```cpp
class Item{
public :

    Item(int a, int b)
        : m_var1{a}, m_var2{b}
    {}
    void some_member_func(){
        auto func = [](){
            std::cout << "member vars :" << m_var1 << "," << m_var2 << std::endl;
        };
        func();
    };
private :
    int m_var1;
    int m_var2;
};
```



```
class Item{
public :

    Item(int a, int b)
        : m_var1{a}, m_var2{b}
    {}
    void some_member_func(){
        auto func = [this](){
            std::cout << "member vars :" << m_var1 << "," << m_var2 << std::endl;
        };
        func();
    };
private :
    int m_var1;
    int m_var2;
};
```

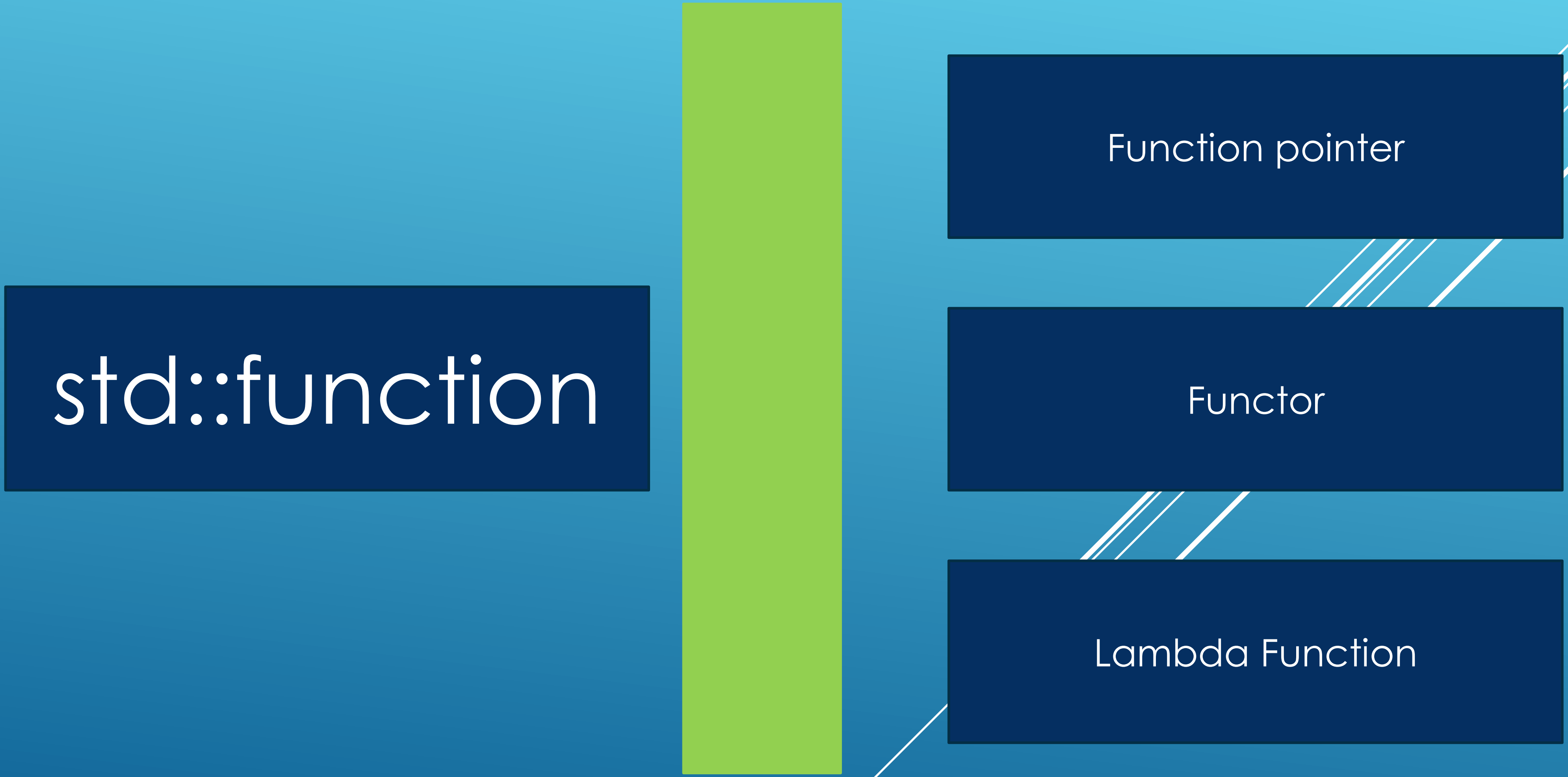


```
class Item{
public :

    Item(int a, int b)
        : m_var1{a}, m_var2{b}
    {}
    void some_member_func(){
        auto func = [=](){
            std::cout << "member vars :" << m_var1 << "," << m_var2 << std::endl;
        };
        func();
    };
private :
    int m_var1;
    int m_var2;
};
```

Slide intentionally left empty

std::function

<functional>



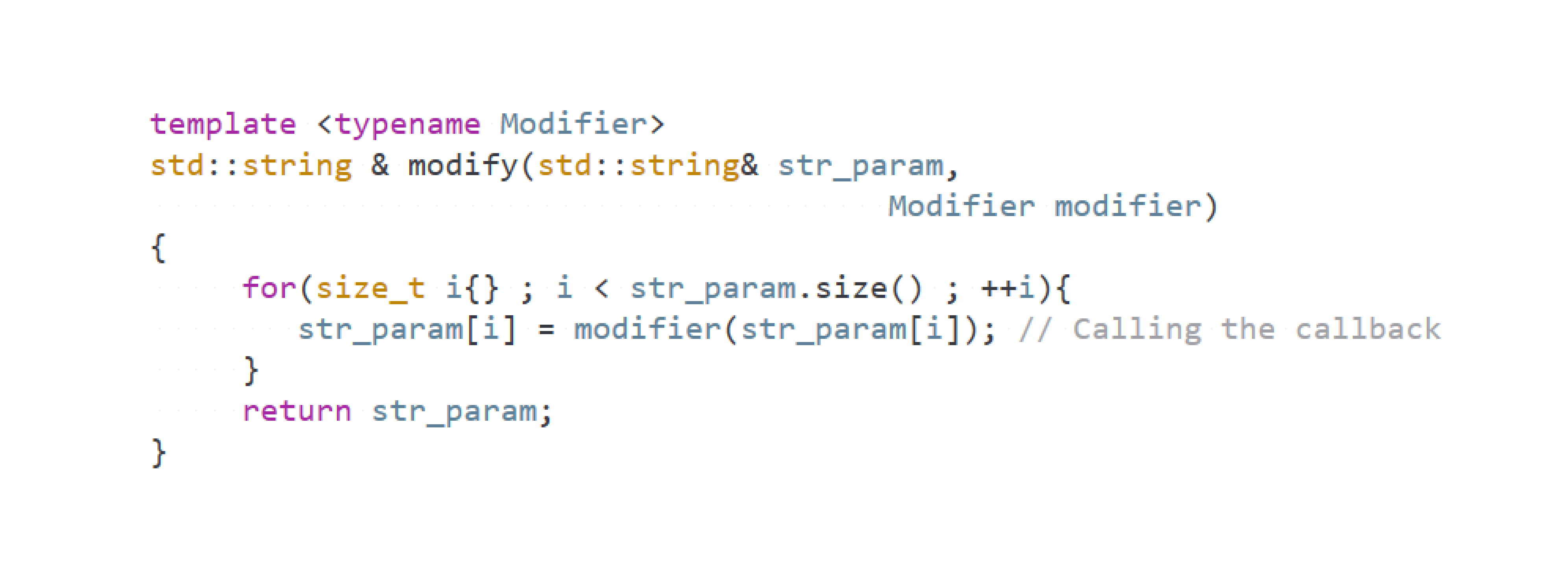
std::function

Function pointer

Functor

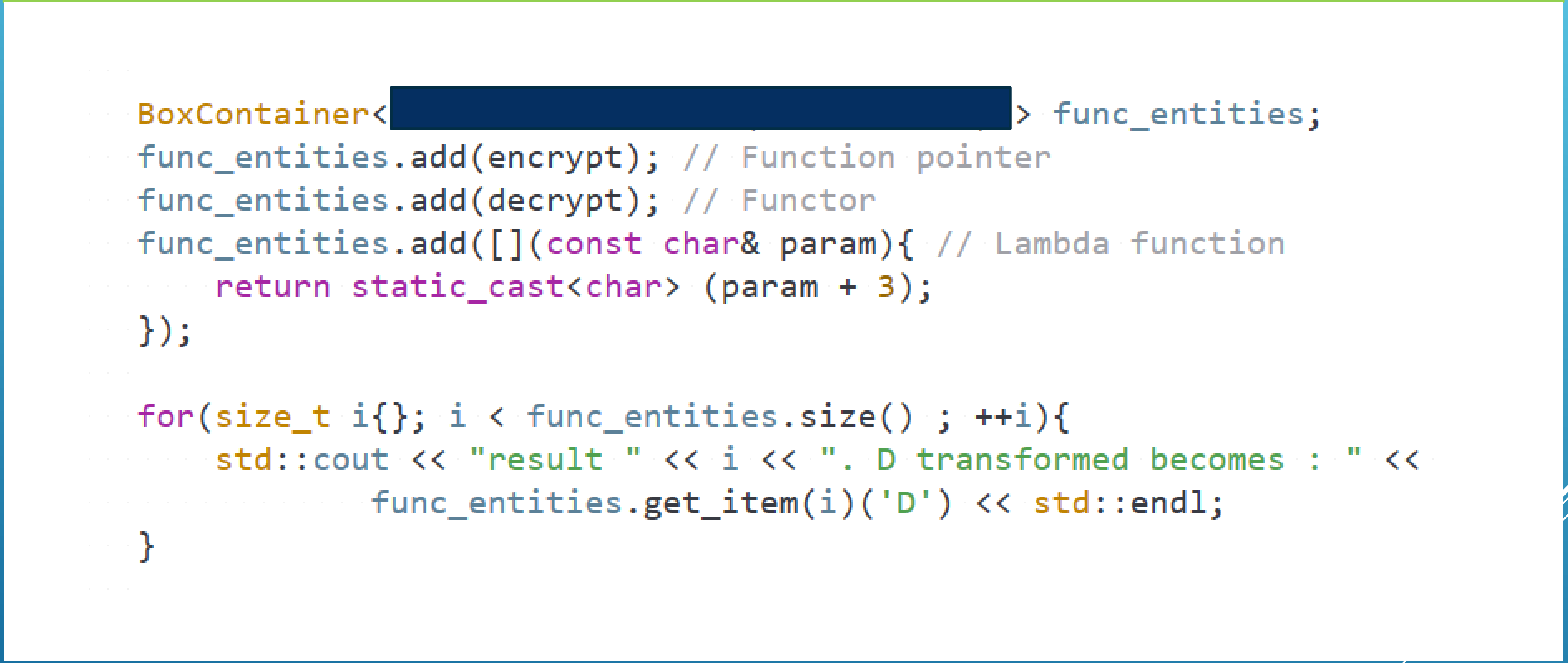
Lambda Function

Abstracting away callback types with templates



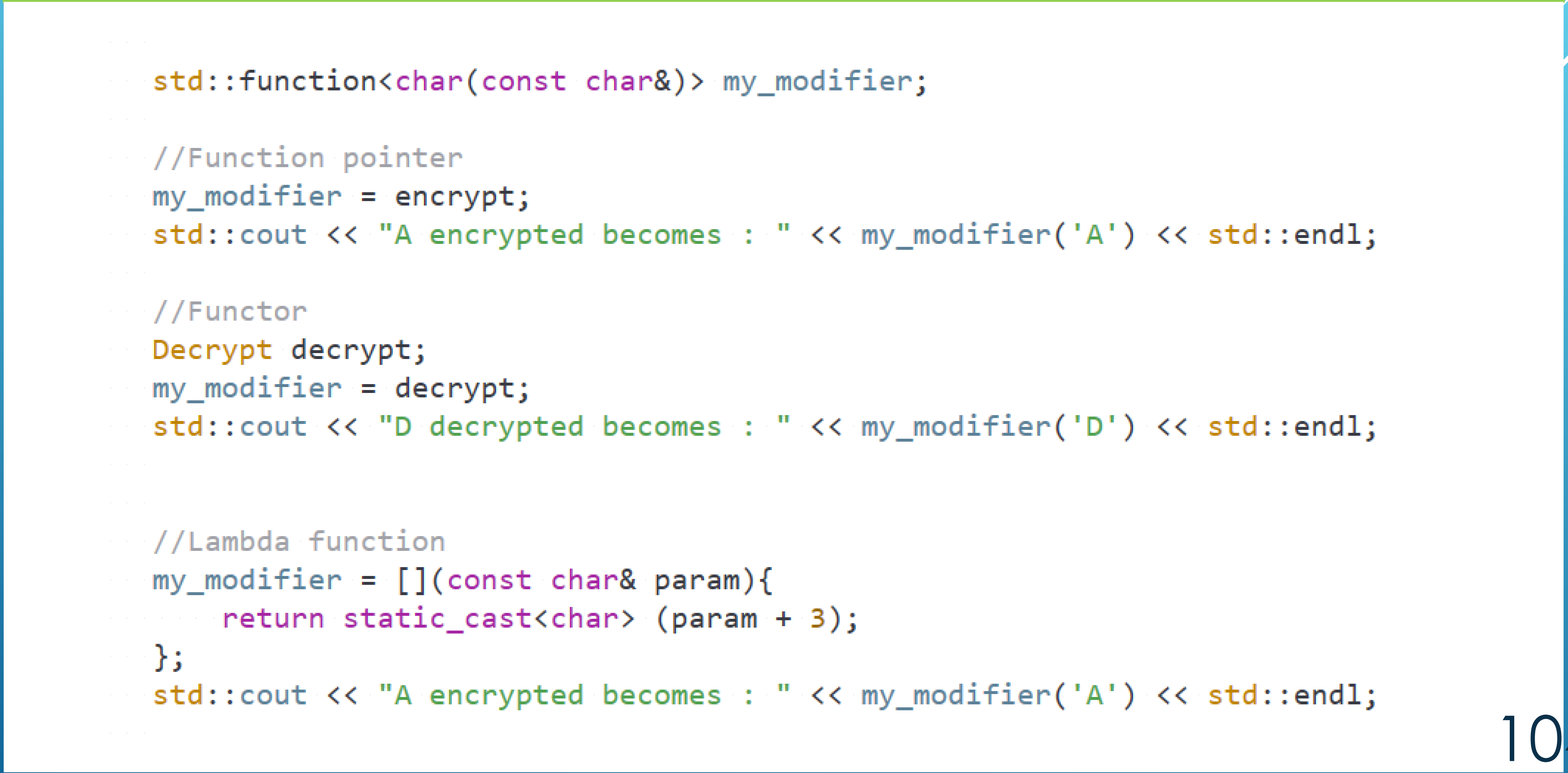
```cpp
template <typename Modifier>
std::string & modify(std::string& str_param,
                                        Modifier modifier)
{
    for(size_t i{} ; i < str_param.size() ; ++i){
        str_param[i] = modifier(str_param[i]); // Calling the callback
    }
    return str_param;
}
```

Storing callbacks in a container



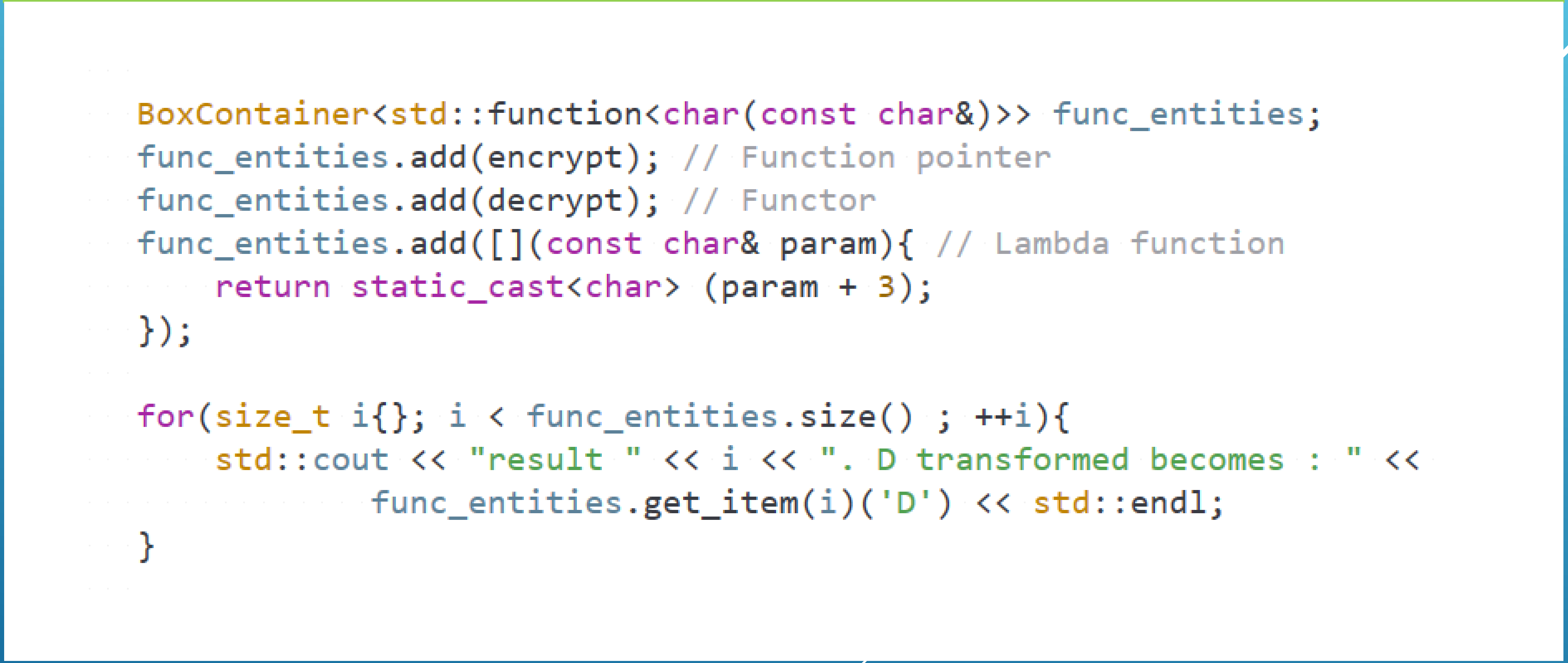
```
BoxContainer<[redacted]> func_entities;
func_entities.add(encrypt); // Function pointer
func_entities.add(decrypt); // Functor
func_entities.add([](const char& param){ // Lambda function
    return static_cast<char> (param + 3);
});

for(size_t i{}; i < func_entities.size() ; ++i){
    std::cout << "result " << i << ". D transformed becomes : " <<
            func_entities.get_item(i)('D') << std::endl;
}
```

Handing function like entities in a unified way



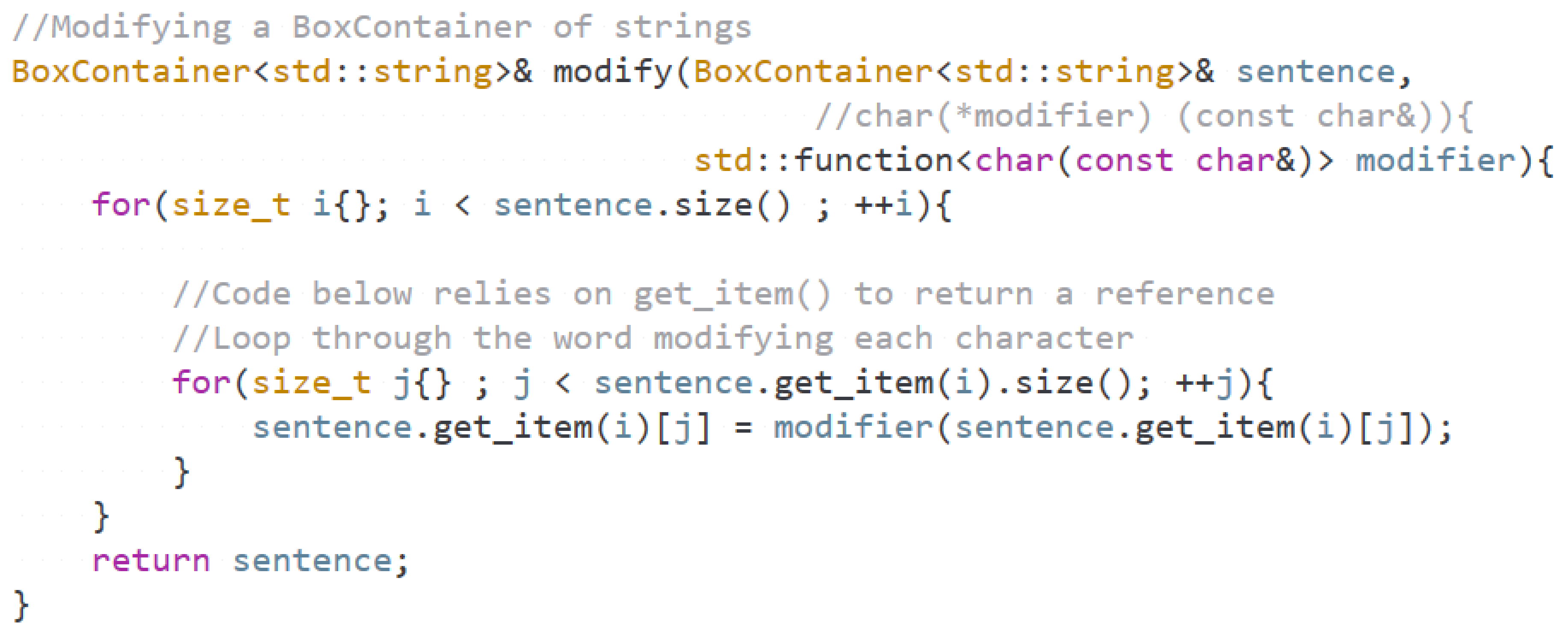
```
std::function<char(const char&)> my_modifier;

//Function pointer
my_modifier = encrypt;
std::cout << "A encrypted becomes : " << my_modifier('A') << std::endl;

//Functor
Decrypt decrypt;
my_modifier = decrypt;
std::cout << "D decrypted becomes : " << my_modifier('D') << std::endl;


//Lambda function
my_modifier = [](const char& param){
    return static_cast<char> (param + 3);
};
std::cout << "A encrypted becomes : " << my_modifier('A') << std::endl;
```

Storing function like entities in collections, like BoxContainer



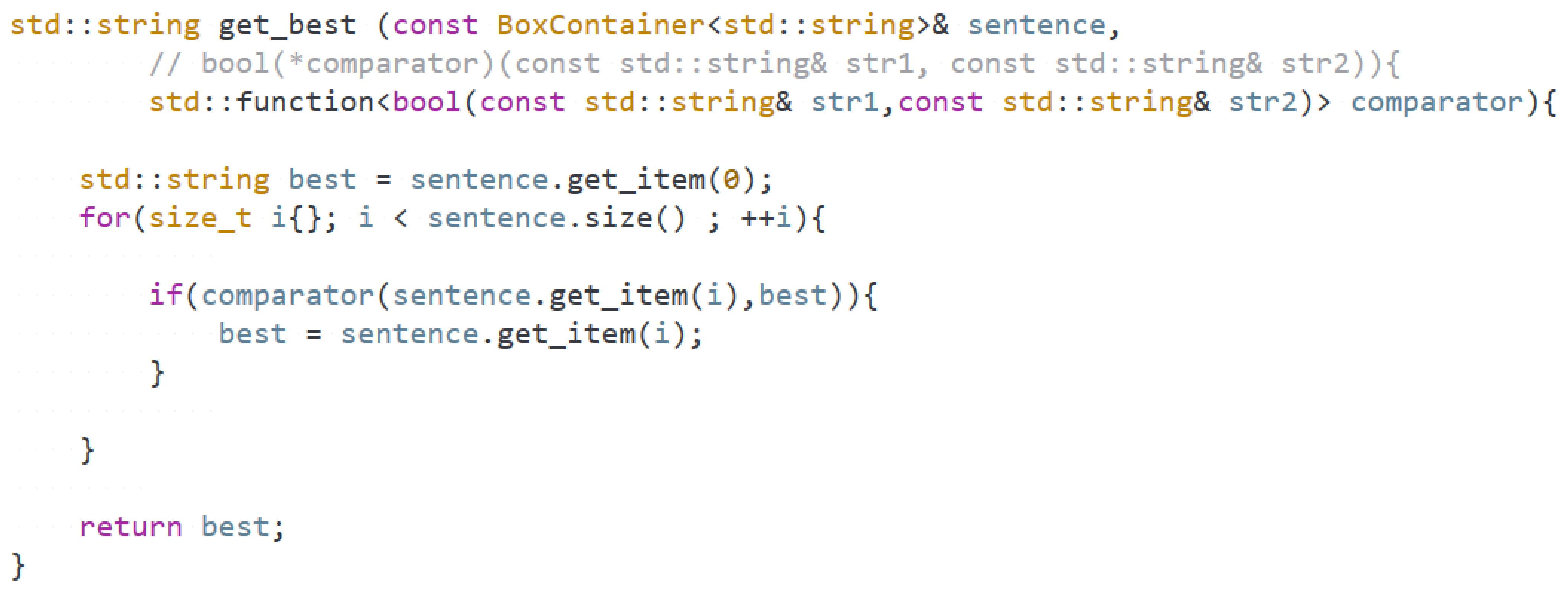
```cpp
BoxContainer<std::function<char(const char&)>> func_entities;
func_entities.add(encrypt); // Function pointer
func_entities.add(decrypt); // Functor
func_entities.add([](const char& param){ // Lambda function
    return static_cast<char> (param + 3);
});

for(size_t i{}; i < func_entities.size() ; ++i){
    std::cout << "result " << i << ". D transformed becomes : " <<
            func_entities.get_item(i)('D') << std::endl;
}
```

Std::function as callback



```
//Modifying a BoxContainer of strings
BoxContainer<std::string>& modify(BoxContainer<std::string>& sentence,
                                        //char(*modifier) (const char&)){
                                  std::function<char(const char&)> modifier){
    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;
}
```

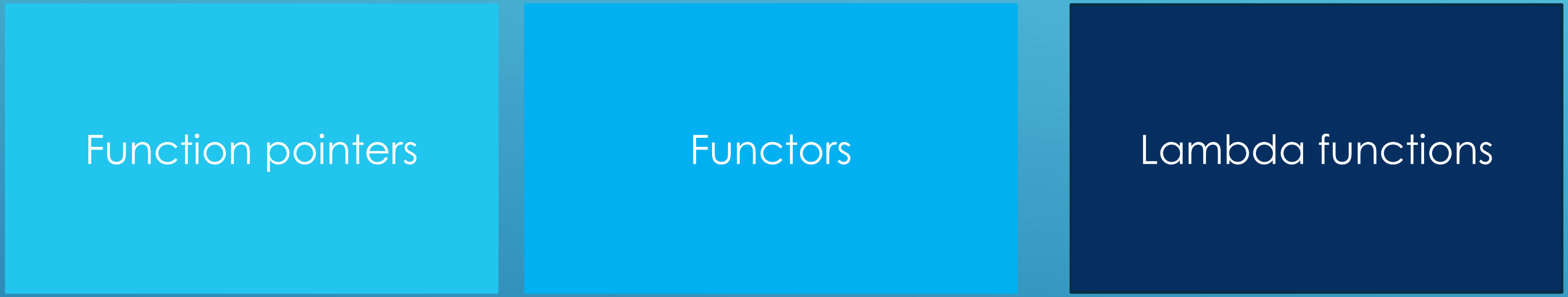
std::function as callback



```
std::string get_best (const BoxContainer<std::string>& sentence,
        // bool(*comparator)(const std::string& str1, const std::string& str2)){
        std::function<bool(const std::string& str1,const std::string& str2)> comparator){

    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;
}
```

Slide intentionally left empty

Function Like entities : Summary

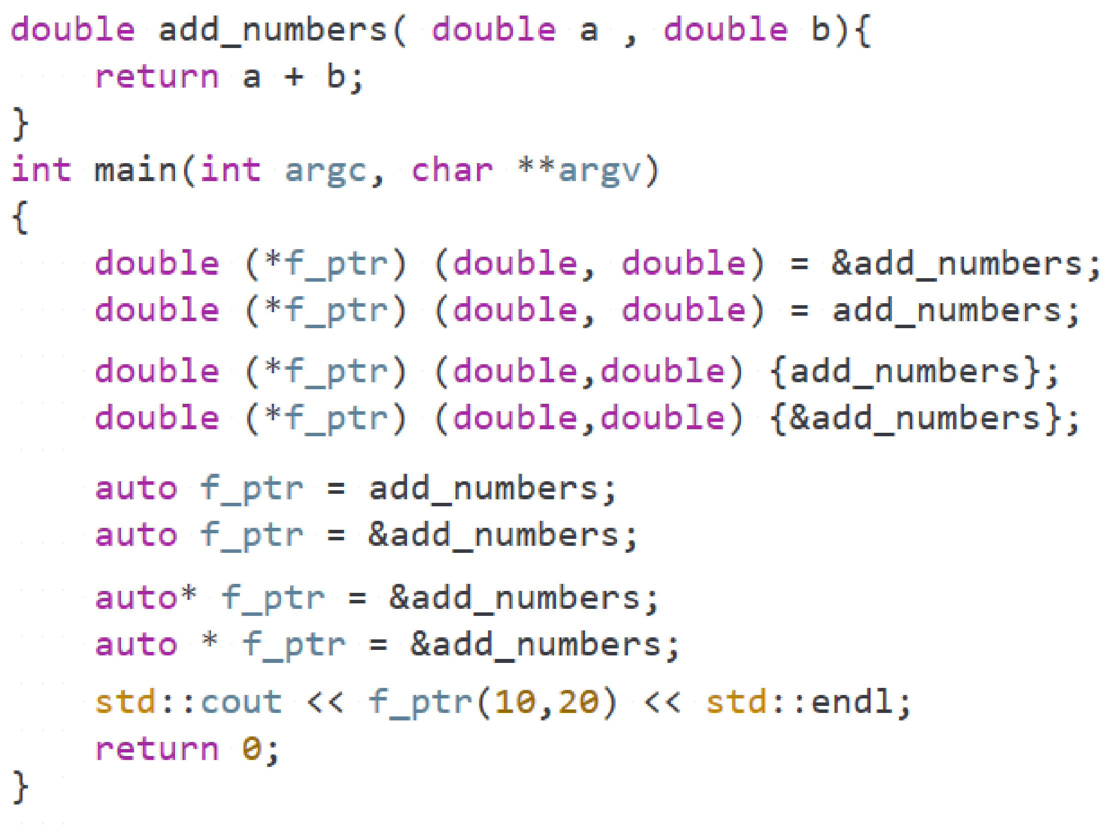


Function pointers

Functors

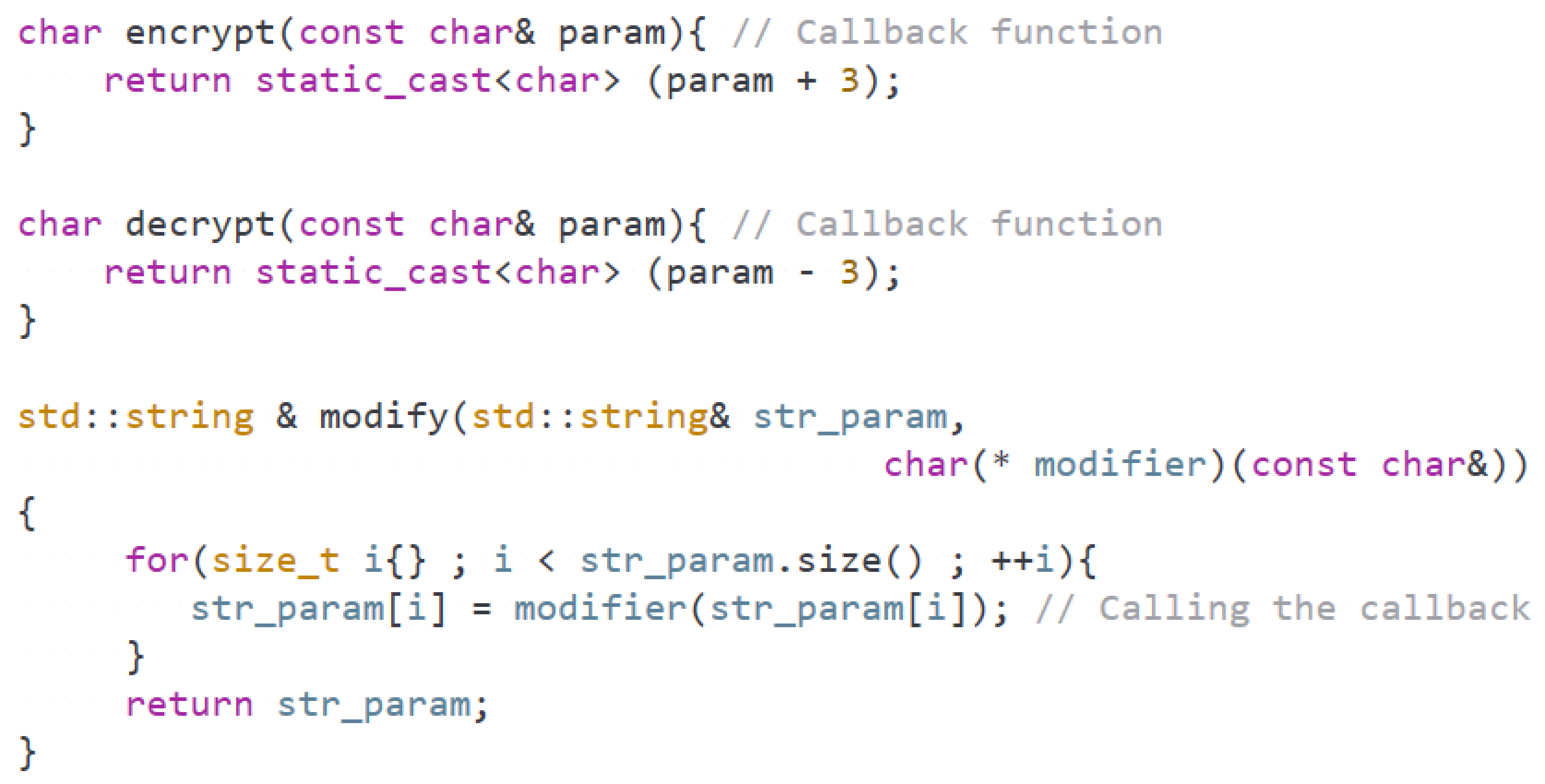
Lambda functions

Function pointers



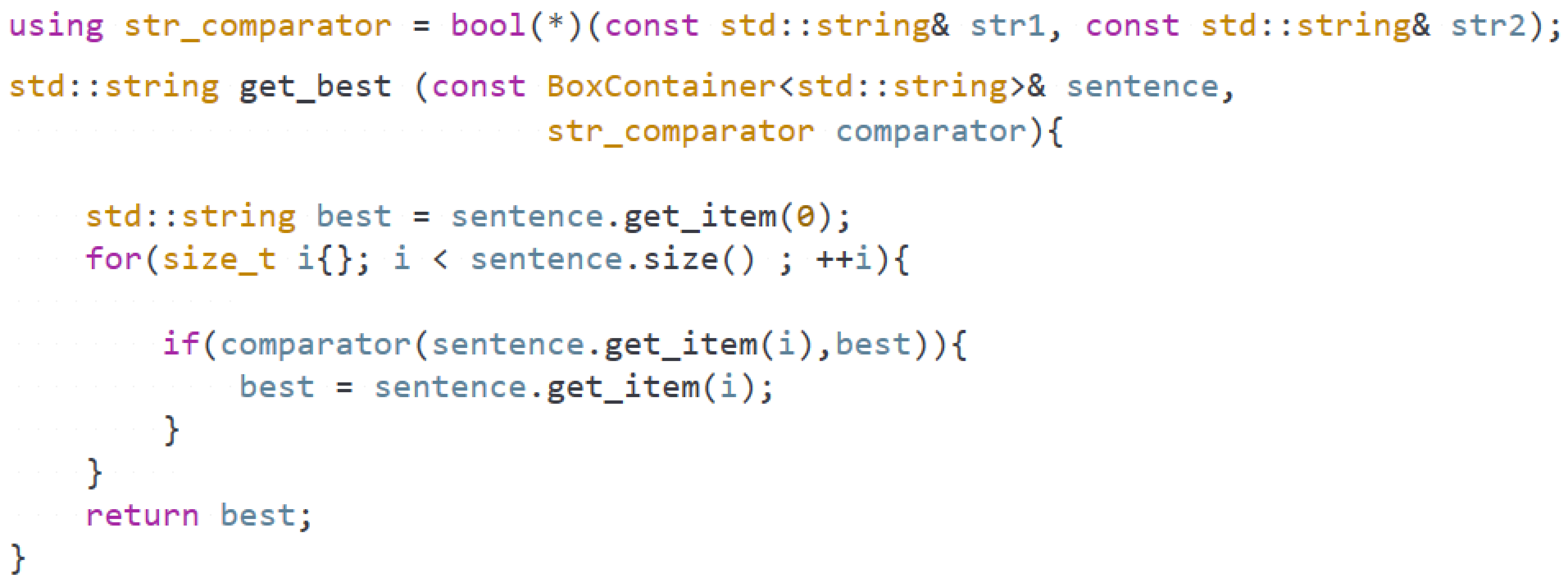
```cpp
double add_numbers( double a , double b){
    return a + b;
}
int main(int argc, char **argv)
{
    double (*f_ptr) (double, double) = &add_numbers;
    double (*f_ptr) (double, double) = add_numbers;

    double (*f_ptr) (double,double) {add_numbers};
    double (*f_ptr) (double,double) {&add_numbers};

    auto f_ptr = add_numbers;
    auto f_ptr = &add_numbers;

    auto* f_ptr = &add_numbers;
    auto * f_ptr = &add_numbers;

    std::cout << f_ptr(10,20) << std::endl;
    return 0;
}
```

Callback functions



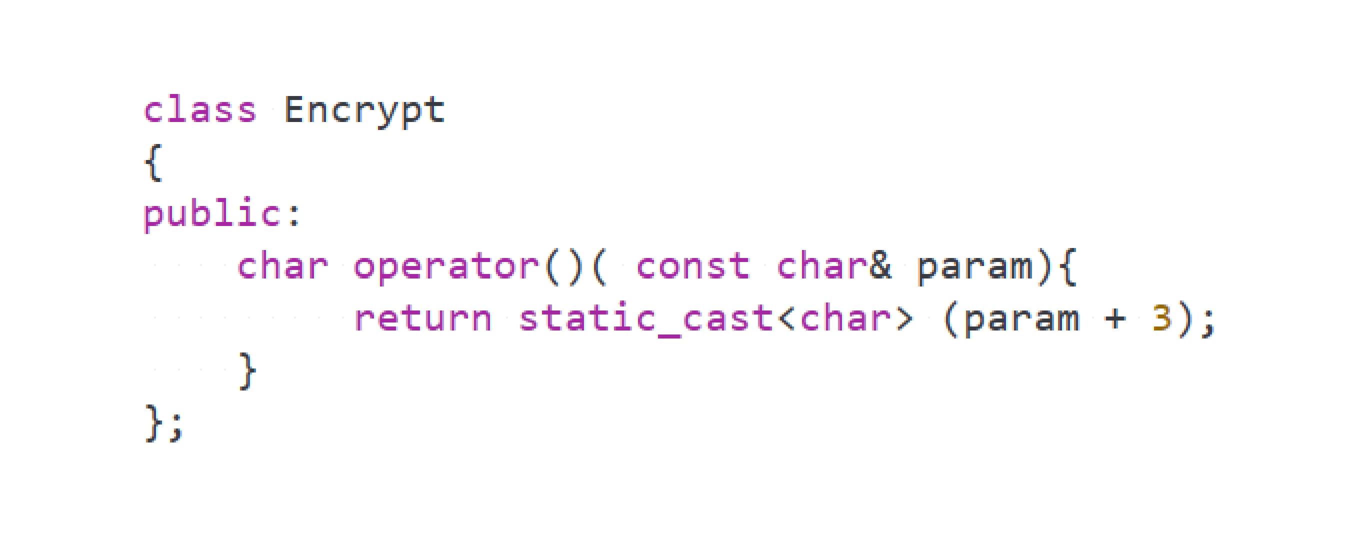
```
char encrypt(const char& param){ // Callback function
    return static_cast<char> (param + 3);
}

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

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;
}
```

Function pointer Type aliases with “using”



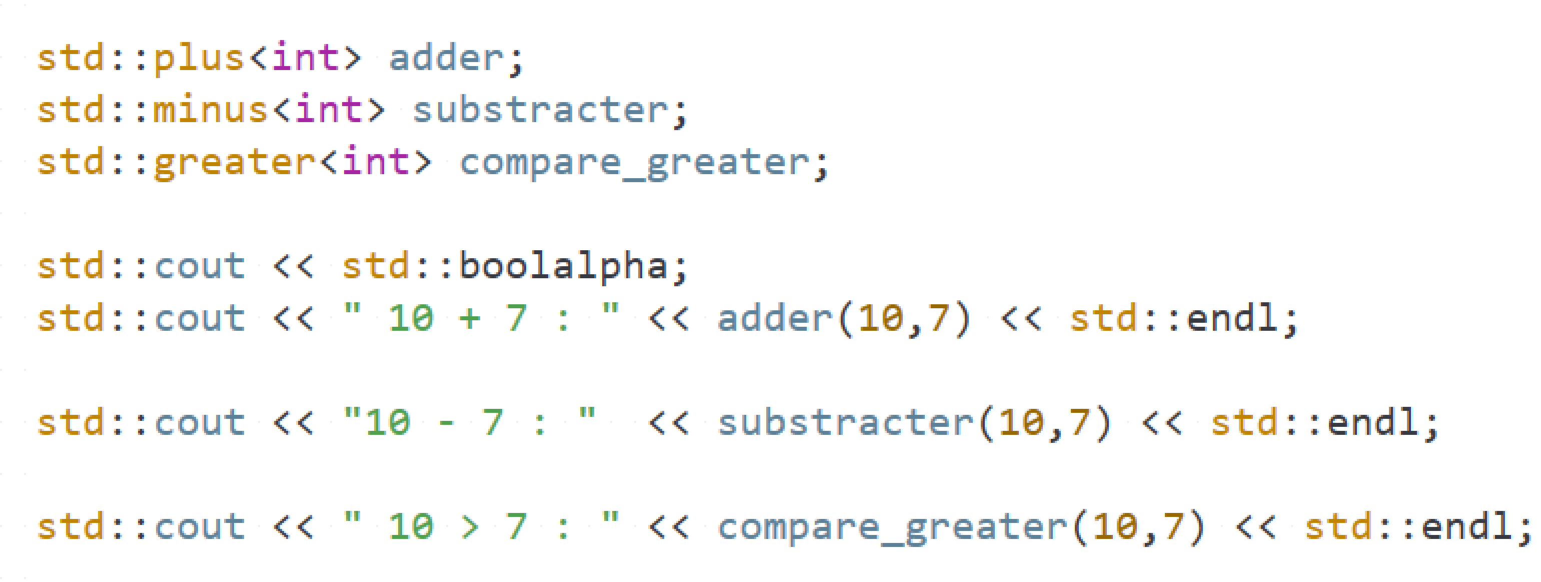
```cpp
using str_comparator = bool(*)(const std::string& str1, const std::string& str2);

std::string get_best (const BoxContainer<std::string>& sentence,
                            str_comparator comparator){

    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;
}
```

Functors



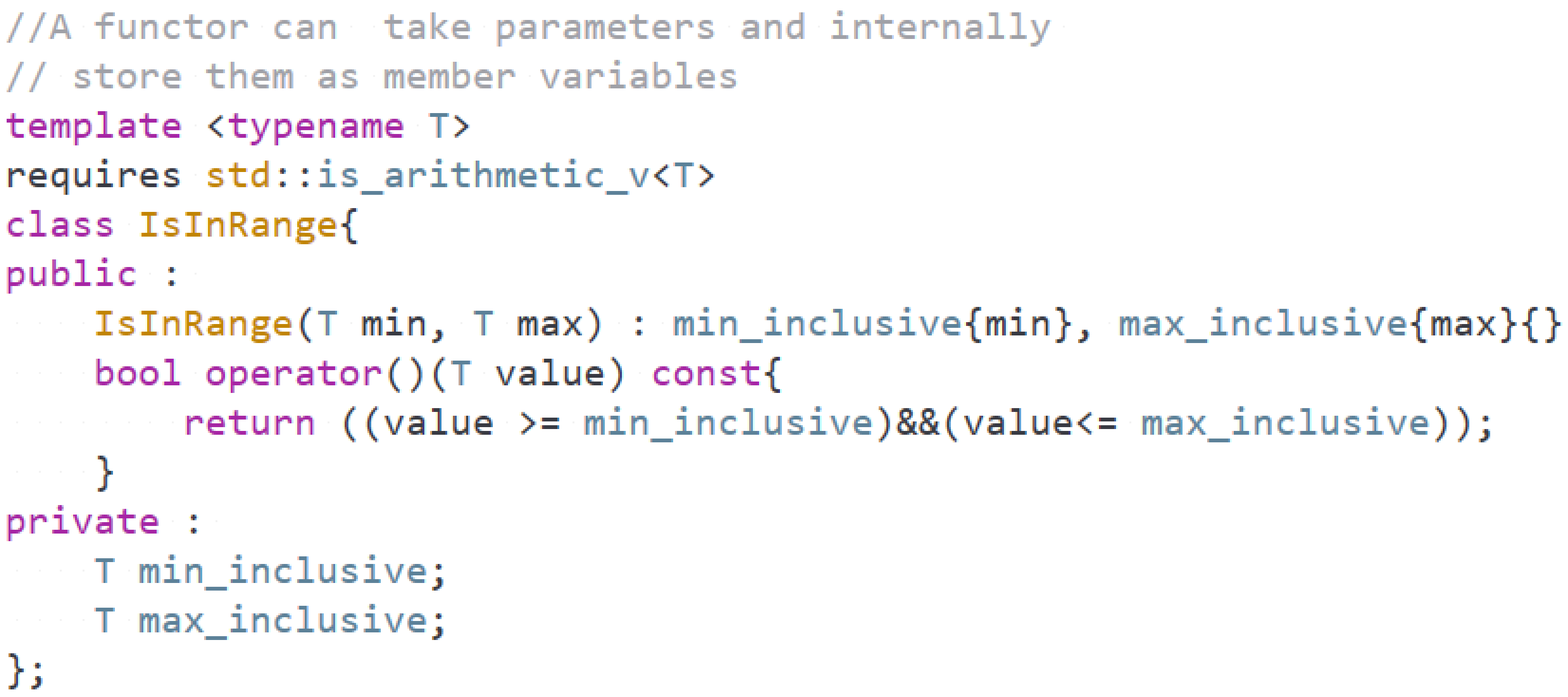
```
class Encrypt
{
public:
    char operator()( const char& param){
        return static_cast<char> (param + 3);
    }
};
```

Standard functors



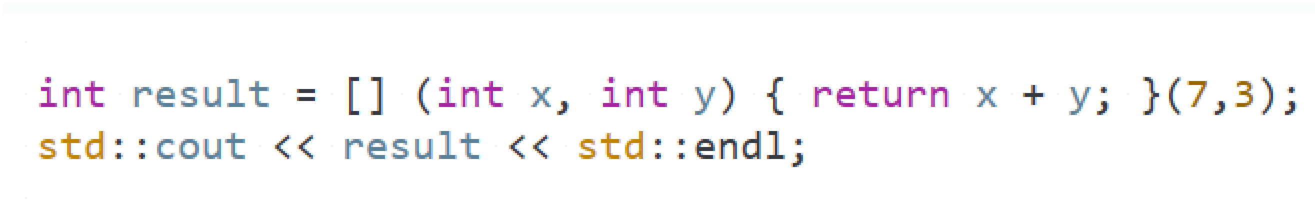
```
std::plus<int> adder;
std::minus<int> substracter;
std::greater<int> compare_greater;

std::cout << std::boolalpha;
std::cout << " 10 + 7 : " << adder(10,7) << std::endl;

std::cout << "10 - 7 : "  << substracter(10,7) << std::endl;

std::cout << " 10 > 7 : " << compare_greater(10,7) << std::endl;
```

Functors with parameters



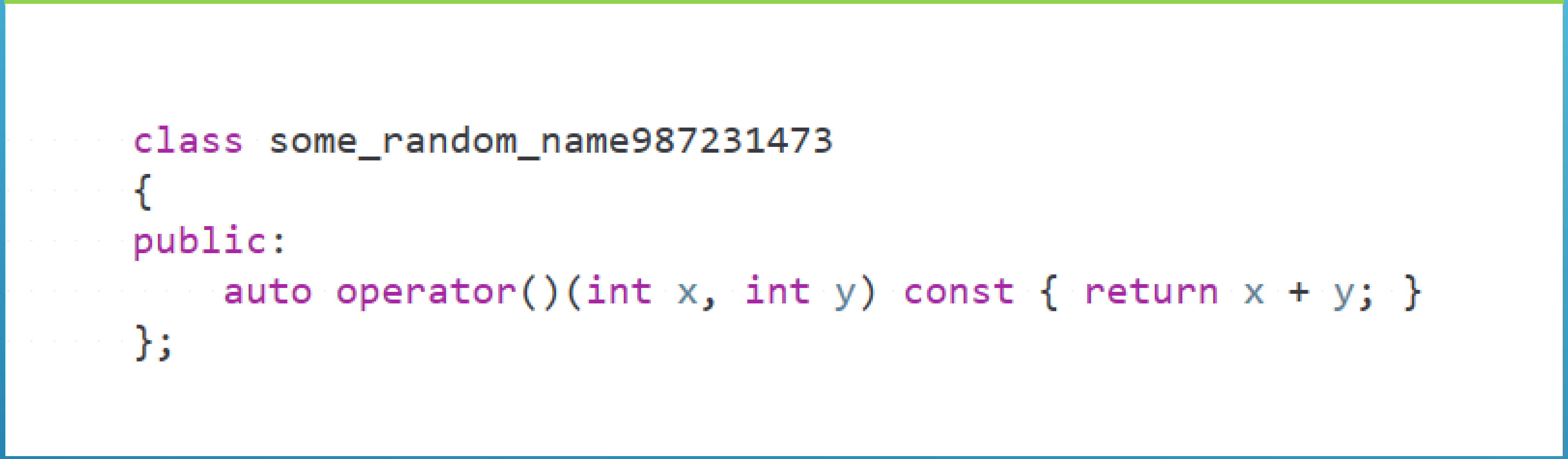
```cpp
//A functor can  take parameters and internally
// store them as member variables
template <typename T>
requires std::is_arithmetic_v<T>
class IsInRange{
public :
    IsInRange(T min, T max) : min_inclusive{min}, max_inclusive{max}{}
    bool operator()(T value) const{
        return ((value >= min_inclusive)&&(value<= max_inclusive));
    }
private :
    T min_inclusive;
    T max_inclusive;
};
```

Lambdas are functors behind the scenes



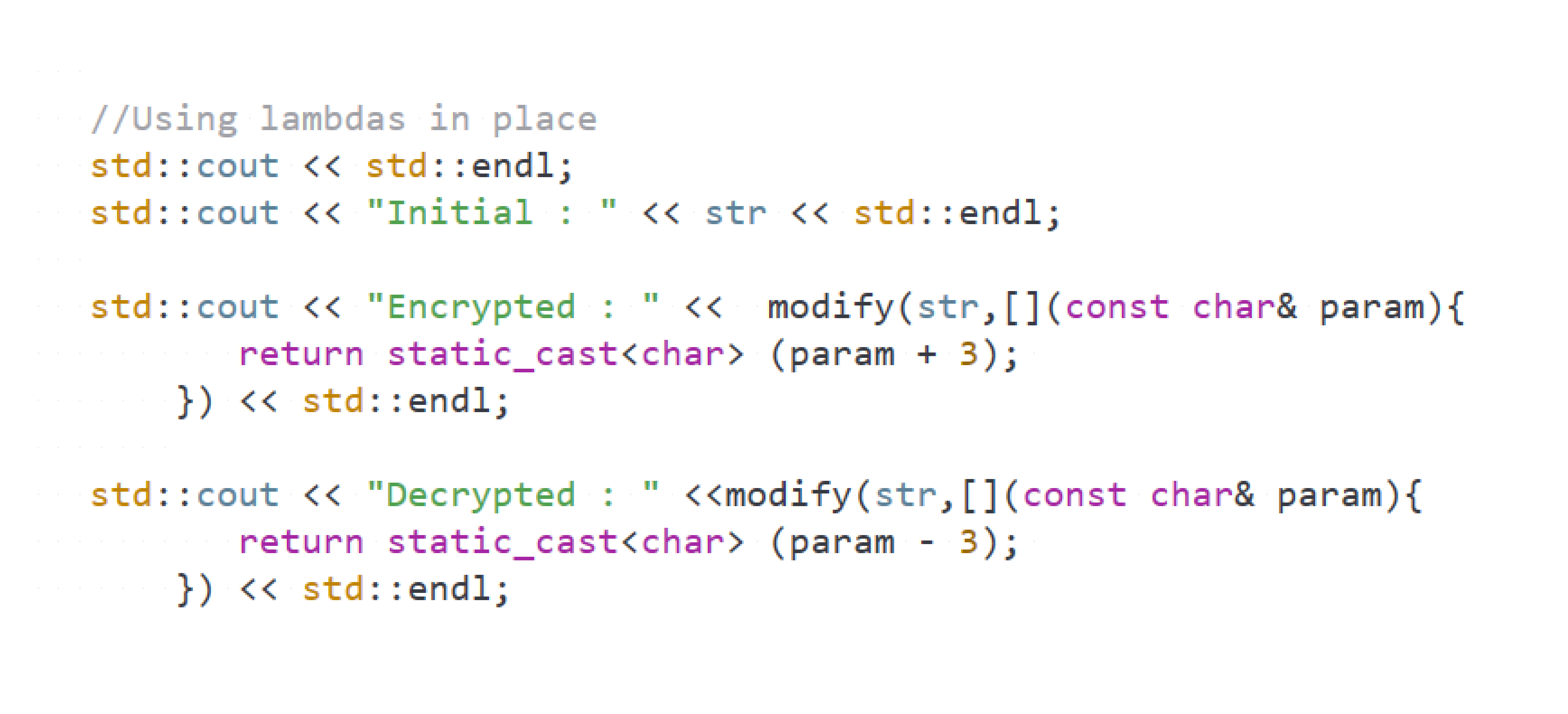
```
int result = [] (int x, int y) { return x + y; }(7,3);
std::cout << result << std::endl;
```

Lambdas are functors behind the scenes



```cpp
class some_random_name987231473
{
public:
    auto operator()(int x, int y) const { return x + y; }
};
```

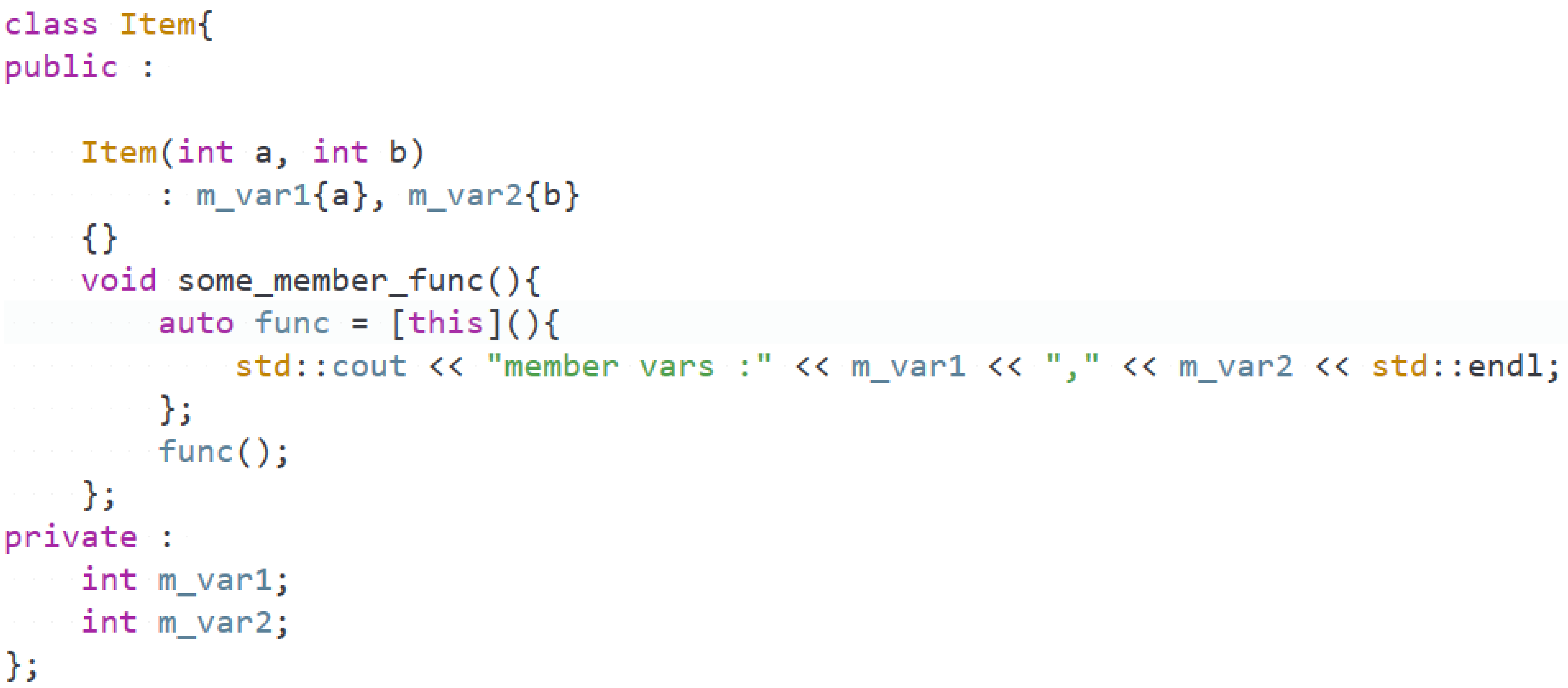
Lambdas as callbacks



```
//Using lambdas in place
std::cout << std::endl;
std::cout << "Initial : " << str << std::endl;

std::cout << "Encrypted : " <<  modify(str,[](const char& param){
       return static_cast<char> (param + 3);
    }) << std::endl;

std::cout << "Decrypted : " <<modify(str,[](const char& param){
       return static_cast<char> (param - 3);
    }) << std::endl;
```

Lambda captures under the hood

Capturing the this pointer



```
class Item{
public :

    Item(int a, int b)
        : m_var1{a}, m_var2{b}
    {}
    void some_member_func(){
        auto func = [this](){
            std::cout << "member vars :" << m_var1 << "," << m_var2 << std::endl;
        };
        func();
    };
private :
    int m_var1;
    int m_var2;
};
```

std::function

- Function pointer
- Functor
- Lambda Function