

MODERN C++ #2

ADVANCED FEATURES



CODERS
SCHOOL

ŁUKASZ ZIOBRÓŃ

KAMIL SZATKOWSKI

AGENDA

- intro (15')
- attributes (20')
- `constexpr` (45')
- ☕ break (15')
- `noexcept` (25')
- data structure alignment (`alignas`, `alignof`) (20')
- structured bindings (30')
- ☕ break (10')
- lambda expressions in short (1h)
- 🍷 lunch break (50')
- other useful features - review only (1h)
- recap (20')

LET'S GET TO KNOW EACH OTHER

- Your name and programming experience
- What you don't like in C++?
- Your hobbies

ŁUKASZ ZIOBRŃ

NOT ONLY A PROGRAMMING XP

- Entrepreneur & CEO @ Coders School
- C++ and Python developer @ Nokia & Credit Suisse
- Team leader & Trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Webdeveloper (HTML, PHP, CSS) @ StarCraft Area

TRAINING EXPERIENCE

- C++ trainings @ Coders School
- Practical Aspects Of Software Engineering @ PWr & UWr
- Nokia Academy @ Nokia
- Internal corporate trainings





PUBLIC SPEAKING EXPERIENCE

- code::dive conference
- code::dive community
- Academic Championships in Team Programming*

HOBBIES

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Andragogy

CONTRACT

-  Vegas rule
-  Discussion, not a lecture
-  Additional breaks on demand
-  Be on time

PRE-TEST

WHICH LAMBDA FUNCTION IS VALID?

1. `[]() -> int { return 4; };`
2. `int [](){ return 4; };`
3. `auto [](){ return 4; };`
4. `[]() -> auto {return 4; };`
5. `[](){ return 4; };`
6. `[] { return 4; }`
7. `[] mutable { return 4; }`
8. `[] -> int { return 4; }`
9. `int []{ return 4; }`

ATTRIBUTES

ATTRIBUTES

Attributes provide the unified standard syntax for implementation-defined language extensions, such as the GNU and IBM language extensions `__attribute__((...))`, Microsoft extension `declspec()`, etc.

STANDARD ATTRIBUTES

- `[[noreturn]]` - function does never return, like `std::terminate`. If it does, we have UB
- `[[deprecated]]` (C++14) - function is deprecated
- `[[deprecated("reason")]]` (C++14) - as above, but compiler will emit the reason
- `[[fallthrough]]` (C++17) - in `switch` statement, indicated that fall through is intentional
- `[[nodiscard]]` (C++17) - you cannot ignore value returned from function
- `[[maybe_unused]]` (C++17) - suppress compiler warning on unused class, typedef, variable, function, etc.

[[noreturn]] ATTRIBUTE

```
[[noreturn]] void f() {  
    throw "error";  
    // OK  
}  
  
[[noreturn]] void q(int i) {  
    if (i > 0) {  
        throw "positive";  
    }  
    // the behavior is undefined if called with argument <=0  
}
```

[[fallthrough]] ATTRIBUTE

```
void f(int n) {  
    void g(), h(), i();  
    switch(n) {  
        case 1:  
        case 2:  
            g();  
            [[fallthrough]];  
        case 3: // no warning on fallthrough  
            h();  
        case 4: // compiler may warn on fallthrough  
            i();  
            [[fallthrough]]; // illformed, not before a case label  
    }  
}
```

[[nodiscard]] ATTRIBUTE

```
struct [[nodiscard]] error_info {};  
error_info process(Data*);  
  
// ...  
  
void passMessage() {  
    auto data = getData();  
    process(data); // compiler warning, discarding error_info  
}
```

[[maybe_unused]] ATTRIBUTES

```
[[maybe_unused]] void f([[maybe_unused]] bool thing1,  
                          [[maybe_unused]] bool thing2)  
{  
    [[maybe_unused]] bool b = thing1 && thing2;  
    assert (b); // in release mode, assert is compiled out, and b is unused  
               // no warning because it is declared [[maybe_unused]]  
} // parameters thing1 and thing2 are not used, no warning
```

[[deprecated]] attribute

Attributes for namespaces and enumerators are available from C++17.

```
[[deprecated("Please use f2 instead")]] int f1();
```

```
enum E {  
    foobar = 0,  
    boobat [[deprecated]] = foobar  
};  
E e = foobat; // Emits warning  
  
namespace [[deprecated]] old_stuff {  
    void legacy();  
}  
old_stuff::legacy(); //Emits warning
```

EXERCISE

Add a new method `double getPi()` in `Circle` class, which returns a PI number. Mark it as deprecated.

constexpr

constexpr KEYWORD

Rationale: faster runtime binary by moving some computations at compile-time.

constexpr is an expression that can be evaluated at compile time and can appear in **constant expressions**. We can have:

- constexpr variable
- constexpr function
- constexpr constructor
- constexpr lambda (default from C++17)
- constexpr if (until C++17)

constexpr VARIABLES

```
int a = 10;           // variable
const int b = 20;     // constant
const double c = 20;  // constant
constexpr int d = 30; // constant at compile-time

constexpr auto e = a; // error: initializer is not a constant expression
constexpr auto f = b; // OK for integral, C++03 compatibility exception
constexpr auto g = c; // error: initializer is not a constant expression
constexpr auto h = d; // OK
```

- `constexpr` variable must be initialized immediately with constant expression. `const` does not need to be initialized with constant expression.
- `constexpr` variable must be a **LiteralType**

constexpr FUNCTIONS

```
constexpr int factorial11(int n) { // C++11 compatible
{
    return (n == 0) ? 1 : n * factorial11(n-1);
}

constexpr int factorial14(int n) { // C++14
    if (n == 0) {
        return 1;
    } else {
        return n * factorial14(n-1);
    }
}
```

constexpr function can be evaluated in both compile time and runtime. Evaluation at compile time can occur when the result is assigned to constexpr variable and arguments can be evaluated at compile time.

`constexpr` FUNCTIONS RESTRICTIONS

In C++11 `constexpr` functions were very restricted - only 1 return instruction (not returning void). From C++14 the only restrictions are, that function must not:

- contain `static` or `thread_local` variables
- contain uninitialized variables
- call non `constexpr` function
- use non-literal types
- be virtual (until C++20)
- use asm code blocks (until C++20)
- have try-catch block or throw exceptions (until C++20)

constexpr CONSTRUCTOR

```
struct Point
{
    constexpr Point(int x, int y)
        : x_(x), y_(y)
    {}

    int x_;
    int y_;
};

constexpr Point a = { 1, 2 };
```

class `Point` can be used in `constexpr` computations, eg in `constexpr` functions. It is a literal type. `constexpr` constructor has the same restrictions as a `constexpr` function and a class cannot have a virtual base class.

constexpr LAMBDA

From C++17 all lambda functions are by default implicitly marked as constexpr, if possible. constexpr keyword can also be used explicitly.

```
auto squared = [](auto x) {                // implicitly constexpr
    return x * x;
};

std::array<int, squared(8)> a;               // OK - array<int, 64>

auto squared = [](auto x) constexpr {     // OK
    return x * x;
};
```

constexpr if

```
if constexpr (a < 0)
    doThis();
else if constexpr (a > 0)
    doThat();
else
    doSomethingElse();
```

`constexpr if` selects only one block of instructions, depending on which condition is met. The condition and other blocks are not compiled in the binary. The condition must be a constant expression.

constexpr if IN SFINAE

constexpr if allows a simplification of template code used by SFINAE idiom.

```
template<class T>    // C++17
auto compute(T x) {
    if constexpr(std::is_scalar_v<T>) {
        return singleComputation(x);
    } else {
        return multipleComputation(x);
    }
}
```

```
template<class T>    // C++11
auto compute(T x) -> enable_if<std::is_scalar<T>::value, int>::type {
    return singleComputation(x);
}
template<class T>
auto compute(T x) -> enable_if<!std::is_scalar<T>::value, int>::type {
    return multipleComputation(x);
}
```


EXERCISE

Write a function that calculates n-th Fibonacci's number. Do not mark it `constexpr`.

In the first line of `main()` add computing 45-th Fibonacci's number. Measure the time of program execution (`time ./modern_cpp`)

Mark fibonacci function as `constexpr`, compile the program and measure the time of execution once again.

If you can't see a big difference assign the result to the `constexpr` variable.

noexcept

noexcept KEYWORD

Rationale: no-throw exception safety guarantee, less code generated for exceptions handling, additional compiler optimisation

Specifies whether a function will throw exceptions or not. If an exception is thrown out of a noexcept function, `std::terminate` is called.

```
void bar() noexcept(true) {}
void baz() noexcept { throw 42; }
// noexcept is the same as noexcept(true)

int main() {
    bar(); // fine
    baz(); // compiles, but calls std::terminate
}
```

noexcept OPERATOR

The `noexcept` operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions. Returns `bool`.

```
void may_throw();  
void no_throw() noexcept;  
  
int main() {  
    std::cout << std::boolalpha  
               << "Is may_throw() noexcept? "  
               << noexcept(may_throw()) << '\n'           // false  
               << "Is no_throw() noexcept? "  
               << noexcept(no_throw()) << '\n';          // true  
}
```

noexcept KEYWORD

Since C++17 exceptions specification is a part of the type system. Below functions are functions of two distinct types:

- `void f() noexcept(true);`
- `void f() noexcept(false);`

This change strengthens the type system, e.g. by allowing APIs to require non-throwing callbacks.

EXERCISE

Mark `getArea()` and `getPerimeter()` methods in `Rectangle` as `noexcept`.

DATA STRUCTURE ALIGNMENT

(alignas, alignof)

alignas KEYWORD

The `alignas` specifier may be applied to:

- the declaration of variable or a class data member
- the declaration or definition of a class/struct/union or enumeration

`alignas(expression)` - expression needs to be positive power of 2.

`alignas(type-id)` - equivalent to `alignas(alignof(type-id))`

`alignas(0)` - has no effect

Exception: if `alignas` would weaken the alignment the type have had without this `alignas`, it will not be applied.

alignas KEYWORD

```
// every object of type see_t will be aligned to 16-byte boundary
struct alignas(16) see_t {
    float see_data[4];
};

// error: requested alignment is not a positive power of 2 alignas(129) char c
alignas(129) char cacheline[128];
```

alignof KEYWORD

The `alignof` specifier returns a value of type `std::size_t`, which is alignment in bytes. If the type is reference type, the operator returns the alignment of referenced type; if the type is array type, alignment requirement of the element type is returned.

```
#include <iostream>
using namespace std;

struct Foo {
    int i;
    float f;
    char c;
};

struct Empty{};
struct alignas(64) Empty64 {};
struct alignas(1) Double {
    double d;
};
```

```
int main() {
    cout << "Alignment of" << '\n'
        << "char: " << alignof(char) << '\n'
        << "pointer: " << alignof(int*) << '\n'
        << "class Foo: " << alignof(Foo) << '\n'
        << "Empty: " << alignof(Empty) << '\n'
        << "Empty64: " << alignof(Empty64) << '\n'
        << "Double: " << alignof(Double) << '\n'
}
```

EXERCISE

Change the alignment of the `Circle` class to 128.

Print the alignment in `main()` function.

Change the alignment to 2.

Print the alignment.

STRUCTURED BINDINGS

STRUCTURED BINDINGS

De-structuring initialization, that allows writting

```
auto [ x, y, z ] = expr;
```

where the type of `expr` is tuple-like objects, whose elements would be bound to the variables `x`, `y` and `z` (which this construct declares).

Tuple-like objects include `std::tuple`, `std::pair`, `std::array` and aggregate structures.

STRUCTURED BINDINGS

```
using Coordinate = std::pair<int, int>;

Coordinate origin() {
    return Coordinate{0, 0};
}

const auto [ x, y ] = origin();
std::cout << "x: " << x << ", y: " << y;
```

```
std::unordered_map<std::string, int> mapping {
    {"a", 1},
    {"b", 2},
    {"c", 3}
};

// De-structure by reference.
for (const auto& [key, value] : mapping) {
    // Do something with key and value
}
```

EXERCISE

Create an `std::map<shared_ptr<Shape>, double>` that will hold a shape and its perimeter.

Use structured bindings to iterate over this collection and display shape info (call `print()` member function) and a perimeter.

LAMBDA EXPRESSIONS IN SHORT

LAMBDA EXPRESSIONS

Rationale: functional programming, in-place functions, more universal function passing

Lambda expressions are defined directly in-place of its usage. Usually it is used as a parameter of another function that expects pointer to function or functor - in general a callable object.

Every lambda expression causes the compiler to create a unique closure class that implements function operator with code from the expression.

Closure is an object of closure class. According to way of capture type this object keeps references or copies of local variables.

BASIC LAMBDA EXPRESSIONS

```
[](){}; // empty lambda  
[] { std::cout << "hello world" << std::endl; } // unnamed lambda  
  
auto l = [] (int x, int y) { return x + y; };  
auto result = l(2, 3); // result = 5
```

LAMBDA'S RETURNED TYPE

From C++14 automatic return type deduction from lambdas works quite good and usually there is no need to tell the returned type directly. However, it can be done using arrow operator.

```
[](bool condition) -> int {  
    if (condition) {  
        return 1;  
    } else {  
        return 2;  
    }  
}
```

PREDICATES

Lambda expressions are usually used to create predicates and functors required by algorithms in standard library (e.g. for `std::sort`).

```
std::array<double, 6> values = { 5.0, 4.0, -1.4, 7.9, -8.22, 0.4 };

std::sort(values.begin(), values.end(), [](double a, double b) {
    return std::abs(a) < std::abs(b); //sort values using
                                     //absolute values
});
```

Output: 0.4, -1.4, 4.0, 5.0, 7.9, -8.22

CAPTURE LIST

Inside `[]` brackets we can include elements that the lambda should capture from the scope in which it is created. Also, the way how they are captured can be specified.

- `[]` empty brackets means that inside the lambda no variable from outer scope can be used.
- `[&]` means that every variable from outer scope is captured by reference, including `this` pointer.
 - Functor created by lambda expression can read and write to any captured variable and all of them are kept inside by lambda reference.
- `[=]` means that every variable from outer scope is capture by value, including `this` pointer.
 - All used variables from the outer scope are copied to lambda expression and can be read only except for `this` pointer.
 - `this` pointer when copied allows lambda to modify all variables it points to.
 - You need a `mutable` keyword to modify values captured by `=`.
- `[capture-list]` allows to explicitly capture variable from the outer scope by mentioning their names on the list.
 - By default all elements are captured by value.
 - If variable should be captured by reference it should be precided by `&` whitch means capturing by reference.
 - Example: `[a, &b]`
- `[*this]` (C++17) captures this pointer by value (creates a copy of this object).

CAPTURE LIST

```
int a {5};
auto add5 = [=] (int x) { return x + a; };

int counter {};
auto inc = [&counter] { counter++; };

int even_count = 0;
for_each(v.begin(), v.end(), [&even_count] (int n) {
    cout << n;
    if (n % 2 == 0)
        ++even_count;
});

cout << "There are " << even_count
    << " even numbers in the vector." << endl;
```

GENERIC LAMBIDAS (C++14)

In C++11 parameters of lambda expression must be declared with use of specific type.

C++14 allows to declare parameter as `auto`.

This allows a compiler to deduce the type of lambda parameter in the same way parameters of the templates are deduced. In result compiler generates a code equivalent to closure class given below:

```
auto lambda = [](auto x, auto y) { return x + y; }

struct UnnamedClosureClass { // code generated by the compiler for above 1 line
    template <typename T1, typename T2>
    auto operator() (T1 x, T2 y) const {
        return x + y;
    }
};

auto lambda = UnnamedClosureClass();
```

LAMBDA CAPTURE EXPRESSIONS (C++14)

C++11 lambda functions capture variables declared in their outer scopes by value-copy or by reference. This means that a value members of a lambda cannot be move-only types.

C++14 allows captured members to be initialized with arbitrary expressions. This allows both capture by value-move and declaring arbitrary members of the lambda, without having a correspondingly named variable in an outer scope.

```
auto lambda = [value = 1] { return value; };  
  
std::unique_ptr<int> ptr(new int(10));  
auto anotherLambda = [value = std::move(ptr)] { return *value; };
```


EXERCISE

Change functions from `main.cpp` into lambdas (`sortByArea`, `perimeterBiggerThan20`, `areaLessThan10`)

Change lambda `areaLessThan10` into lambda `areaLessThanX`, which takes `x = 10` on a capture list. What is the problem?

Use `std::function` to solve the problem.

OTHER USEFUL FEATURES

NESTED NAMESPACE DEFINITIONS (C++17)

You can nest namespaces like this:

```
namespace A::B::C {  
    ...  
}
```

Instead of this:

```
namespace A {  
    namespace B {  
        namespace C {  
            ...  
        }  
    }  
}
```

CLASS TEMPLATE ARGUMENT DEDUCTION (C++17)

From C++17 class template arguments can be deduced automatically. Automatic template argument deduction was available earlier only for template functions.

```
std::pair p(1, 'x'); // C++17: OK, C++14: error: missing
                      //template arguments before p
std::pair<int, std::string> p(1, 'x'); // C++14: OK
auto p = std::make_pair(1, 'x'); // C++17: OK, C++14: OK
```

SELECTION STATEMENTS WITH INITIALIZER (C++17)

New versions of the `if` and `switch` statements for C++:

`if (init; condition)`

```
status_code foo() { // C++14
    { //variable c scope
        status_code c = bar();
        if (c != SUCCESS) {
            return c;
        }
    }
    // ...
}
```

switch (init; condition)

```
status_code foo() { // C++17
    if (status_code c = bar(); c != SUCCESS) {
        return c;
    }
    // ...
}
```

SELECTION STATEMENTS WITH INITIALIZER (C++17)

```
{  
    Foo gadget(args);  
    switch (auto s = gadget.status()) { // C++14  
        case OK: gadget.zip(); break;  
        case Bad: throw BadFoo(s.message());  
    }  
}
```

```
switch (Foo gadget(args); auto s = gadget.status()) { // C++17  
    case OK: gadget.zip(); break;  
    case Bad: throw BadFoo(s.message());  
}
```

OVERVIEW

Overview of modern C++ features

RECAP

**WHAT DO YOU REMEMBER FROM TODAY'S
SESSION?**

C++ QUIRKS

- Lambda - you need to add `mutable` in case you have `[=]` on capture list and you want to modify captured elements
- Lambda - `unique_ptr` on capture list `a=std::move(a)`
- Try marking as many functions as `constexpr` as possible

PRE-TEST

ANSWERS

WHICH LAMBDA FUNCTION IS VALID?

1. `[]() -> int { return 4; };`
2. `int [](){ return 4; };`
3. `auto [](){ return 4; };`
4. `[]() -> auto {return 4; };`
5. `[](){ return 4; };`
6. `[] { return 4; }`
7. `[] mutable { return 4; }`
8. `[] -> int { return 4; }`
9. `int []{ return 4; }`

POST-TEST

Link to post-test will be sent to you in a next week :)

FEEDBACK

- What could be improved in this training?
- What was the most valuable for you?

Training evaluation



COODEERS
THANK YOU 😊
SCHOOL