LAMBDA EXPRESSIONS

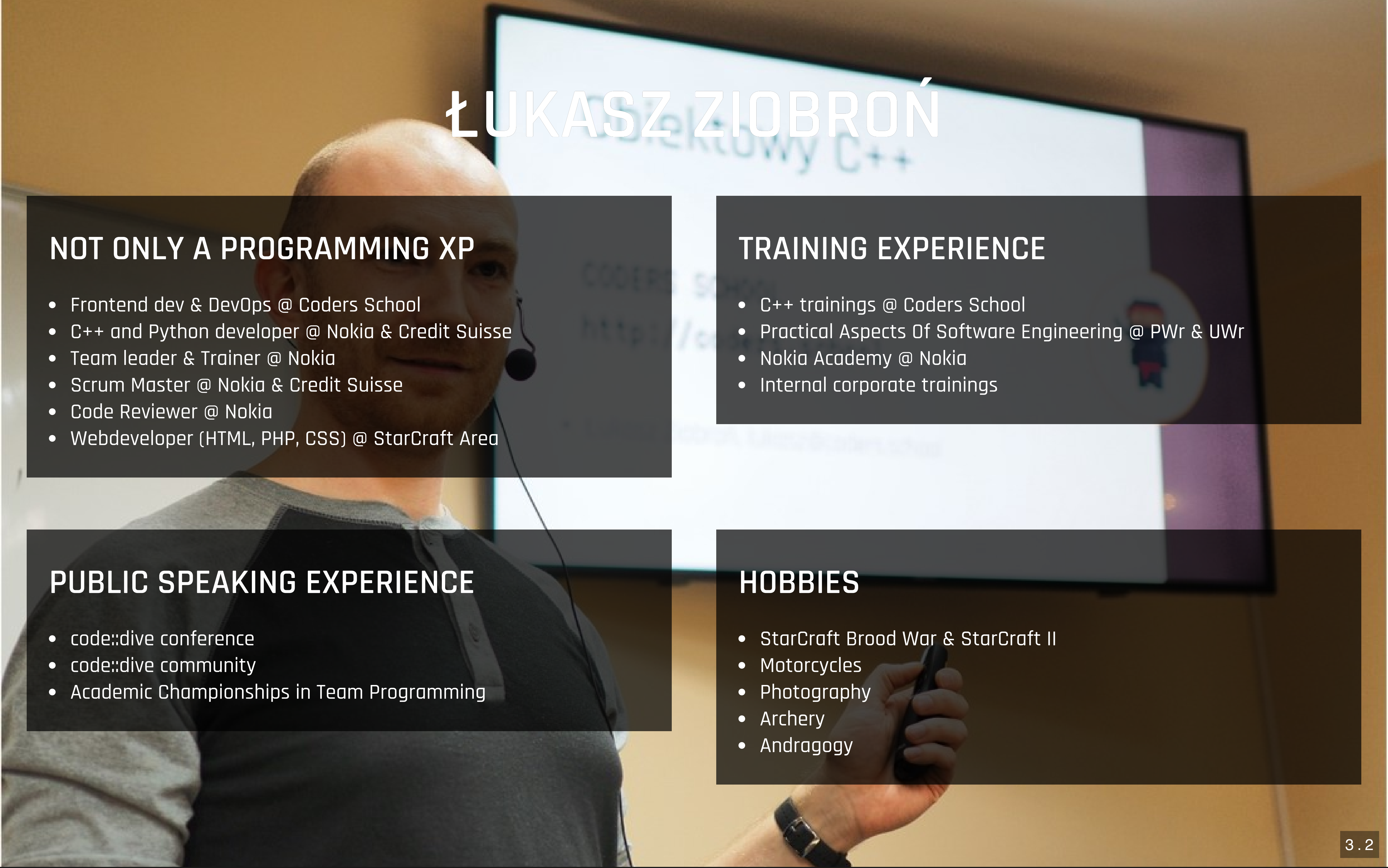
ŁUKASZ ZIOBROŃ

AGENDA

- intro (20")
- STL algorithms customization (30")
- anatomy of lambda (30")
- ☕ break (15")
- capture list and mutable lambdas (30")
- `std::function` vs pointer to function (60")
- ☕ break (10")
- lifetime issues (30")
- `constexpr` lambdas (20")
- 🍝 lunch break (50")
- generic lambdas (10")
- template vs generic lambdas (40")
- recap (20")

LET'S CHECK YOUR MICROPHONE 🙂

- Have you ever used lambda expressions?
- What was the craziest or the most difficult stuff that you have done in C++?



ŁUKASZ ZIOBROŃ

NOT ONLY A PROGRAMMING XP

- Frontend dev & DevOps @ 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 after breaks

LINK TO PRESENTATION ON GITHUB

PRE-TEST

1. 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; }`

2. WHICH CAPTURE LISTS ARE CORRECTLY FORMED?

Assume that all variables exist.

1. `[=, this]`
2. `[&, this]`
3. `[this, *this]`
4. `[&, &a]`
5. `[a, &]`
6. `[&, a, &b]`
7. `[=, a, &b]`
8. `[=, &a]`
9. `[=, *a]`

STL ALGORITHMS CUSTOMIZATION

HOW TO SORT A CONTAINER?

```
std::vector<double> diffs = {1.1, -0.2, -1.3, 0.8, 0.1, -0.3}
```

USE `std::sort`

```
std::sort(diffs.begin(), diffs.end());
```

By default, sorting is done using `std::less` function, which calls `operator<` to compare elements.

RESULT

```
{-1.3, -0.3, -0.2, 0.1, 0.8, 1.1}
```

HOW TO SORT THE CONTAINER BY ABSOLUTE VALUES?

Most of STL algorithms can be customized by passing an additional parameter - a special function object.

```
bool absoluteCompare(double rhs, double lhs) {
    return std::abs(rhs) < std::abs(lhs);
}
std::sort(diffs.begin(), diffs.end(), absoluteCompare);
```

RESULT

```
{0.1, -0.2, -0.3, 0.8, 1.1, -1.3}
```

HOW TO GET TO KNOW WHAT SIGNATURE SHOULD THE FUNCTION OBJECT HAVE?

CUSTOMIZATION FUNCTION

Each algorithm has a documented signature of needed function.

Check `std::sort` on cppreference.com

COMMON KINDS OF FUNCTIONS

- (unary) predicate
  - returns `bool`, takes 1 arg
- binary predicate
  - returns `bool`, takes 2 args
- comparator
  - returns `bool` takes 2 args

WHAT IS A *FUNCTION OBJECT*?

It is any object for which the function call operator is defined - `operator()`.

- pointer to function
- functor
- lambda expression
- `std::function`
- function?
  - no, it's not a *function object*, but it is implicitly convertible to *pointer to function*

SORTING WITH A FUNCTION

```cpp
bool absoluteCompare(double rhs, double lhs) {
    return std::abs(rhs) < std::abs(lhs);
}
std::sort(diffs.begin(), diffs.end(), absoluteCompare);
```

SORTING WITH A FUNCTOR

```cpp
struct AbsoluteCompare {
    bool operator()(double rhs, double lhs) {
        return std::abs(rhs) < std::abs(lhs);
    }
};
std::sort(diffs.begin(), diffs.end(), AbsoluteCompare{});
```

SORTING WITH LAMBDA

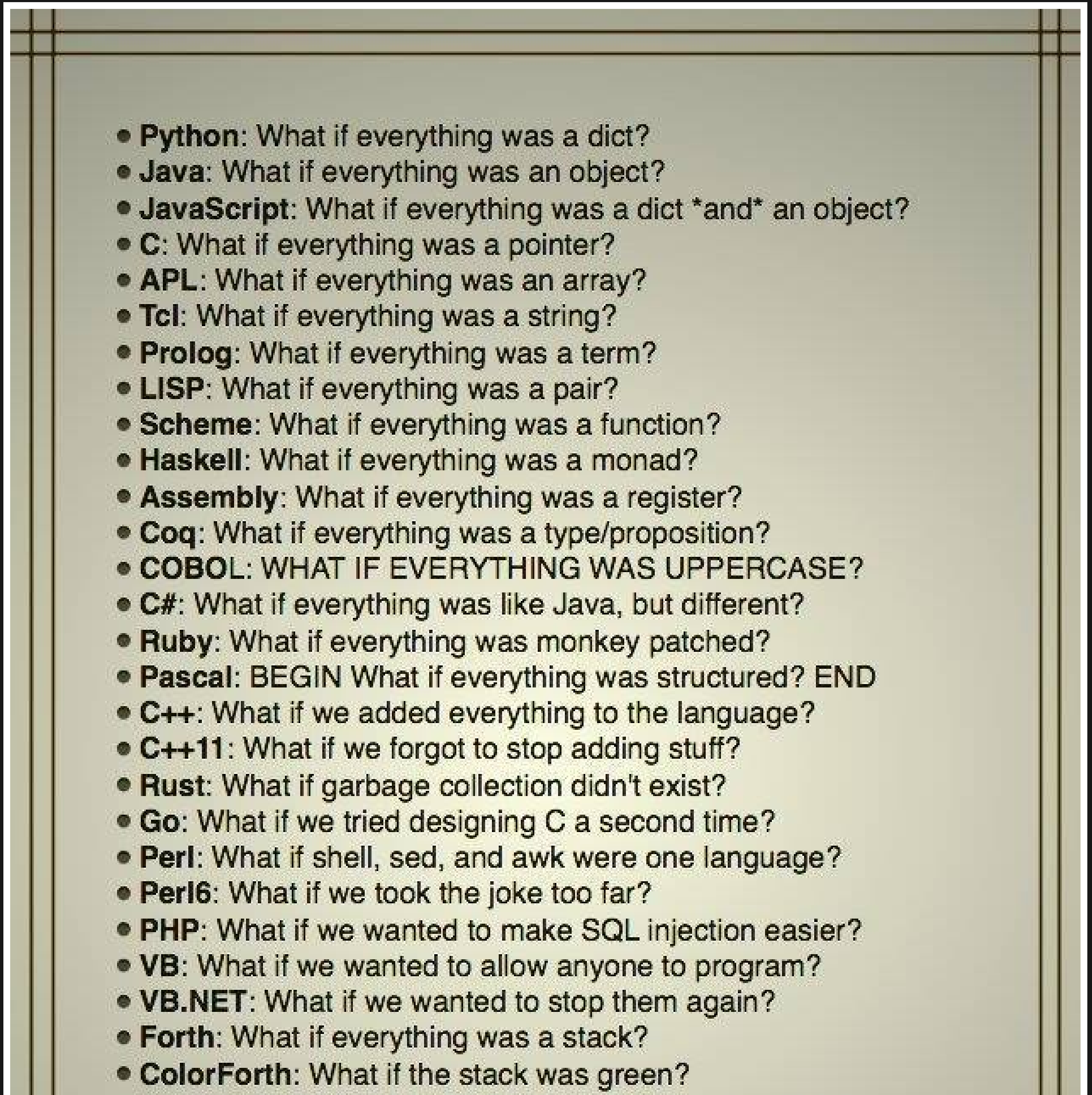
NAMED LAMBDA

```cpp
auto absoluteCompare = [](double rhs, double lhs) {
    return std::abs(rhs) < std::abs(lhs);
};
std::sort(diffs.begin(), diffs.end(), absoluteCompare);
```

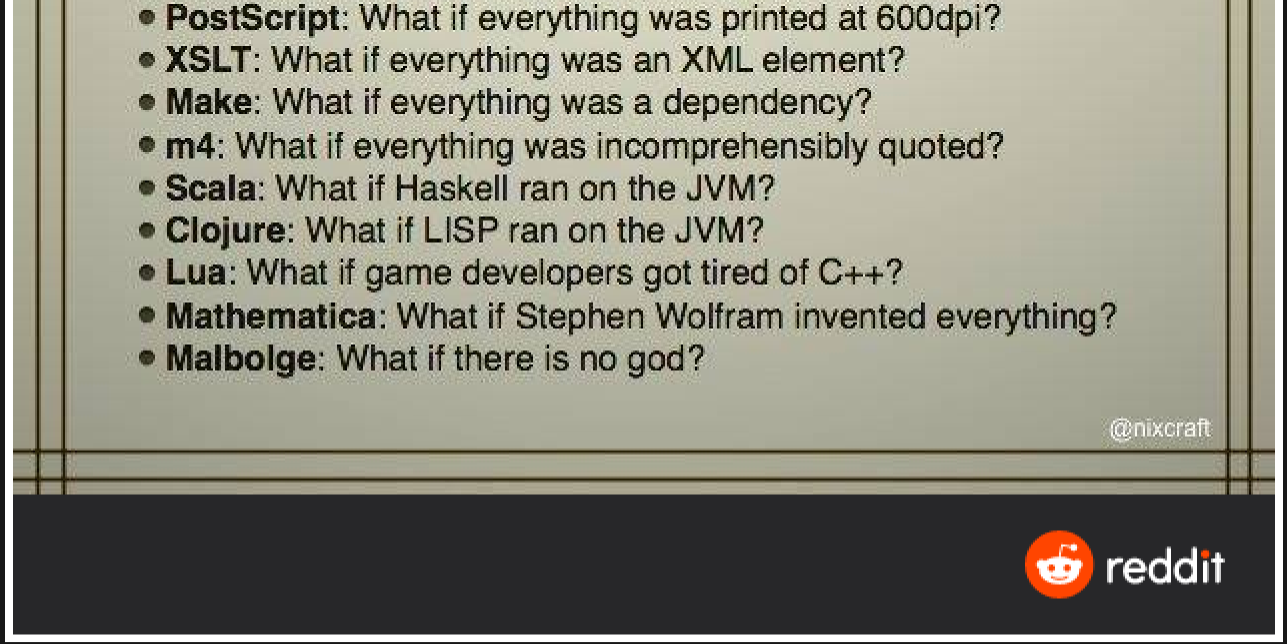
UNNAMED LAMBDA

```cpp
std::sort(diffs.begin(), diffs.end(), [](double rhs, double lhs) {
    return std::abs(rhs) < std::abs(lhs);
});
```

WHY SO MANY WAYS?



- **Python**: What if everything was a dict?
- **Java**: What if everything was an object?
- **JavaScript**: What if everything was a dict *and* an object?
- **C**: What if everything was a pointer?
- **APL**: What if everything was an array?
- **Tcl**: What if everything was a string?
- **Prolog**: What if everything was a term?
- **LISP**: What if everything was a pair?
- **Scheme**: What if everything was a function?
- **Haskell**: What if everything was a monad?
- **Assembly**: What if everything was a register?
- **Coq**: What if everything was a type/proposition?
- **COBOL**: WHAT IF EVERYTHING WAS UPPERCASE?
- **C#**: What if everything was like Java, but different?
- **Ruby**: What if everything was monkey patched?
- **Pascal**: BEGIN What if everything was structured? END
- **C++**: What if we added everything to the language?
- **C++11**: What if we forgot to stop adding stuff?
- **Rust**: What if garbage collection didn't exist?
- **Go**: What if we tried designing C a second time?
- **Perl**: What if shell, sed, and awk were one language?
- **Perl6**: What if we took the joke too far?
- **PHP**: What if we wanted to make SQL injection easier?
- **VB**: What if we wanted to allow anyone to program?
- **VB.NET**: What if we wanted to stop them again?
- **Forth**: What if everything was a stack?
- **ColorForth**: What if the stack was green?



- **PostScript**: What if everything was printed at 600dpi?
- **XSLT**: What if everything was an XML element?
- **Make**: What if everything was a dependency?
- **m4**: What if everything was incomprehensibly quoted?
- **Scala**: What if Haskell ran on the JVM?
- **Clojure**: What if LISP ran on the JVM?
- **Lua**: What if game developers got tired of C++?
- **Mathematica**: What if Stephen Wolfram invented everything?
- **Malbolge**: What if there is no god?

@nixcraft

LAMBDA EXPRESSIONS

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

Lambda expression is usually 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.

EXERCISE

`01_threeWays.cpp`

Use proper STL algorithm to check if all elements of the vector *numbers* are divisible by 3.

```
std::vector numbers = {18, 21, 36, 90, 27, 14, 103};
```

Implement 3 versions:

- lambda
- functor
- function

SOLUTIONS

LAMBDA

```
bool result = std::all_of(numbers.begin(), numbers.end(), [](int number){
    return number % 3 == 0;
});
```

FUNCTION

```
bool isDivisibleBy3(int number) {
    return number % 3 == 0;
}
bool result = std::all_of(numbers.begin(), numbers.end(), isDivisibleBy3);
```

SOLUTIONS

FUNCTOR

```
struct DivisibleBy {
    DivisibleBy(int n) : n_{n} {}
    bool operator()(int number) {
            return number % n_ == 0;
    }
private:
    int n_;
};
bool result = std::all_of(numbers.begin(), numbers.end(), DivisibleBy{3});
```

YOUR CONCLUSIONS

WHAT IS A LAMBDA EXPRESSION?

- It is a function object
- Simple and short to write - `[](){}`
- It is used to have a short form of writing a function object, which normally would take more characters to be typed
- Provides better code readability
- The type of lambda is called "closure class"
- Closure is known only to the compiler
- To assign a lambda to a variable, it's type must be `auto`

THE ANATOMY OF LAMBDA

But first, let's take a look into...

EMPTY FUNCTION

```
void f(){}
```

- `void` - return type
- `f` - function name
- `( )` - empty parameter list
- `{ }` - empty function body

EMPTY LAMBDA

```
[](){};
```

- `[ ]` - capture list (which variables from the outer scope will be captured)
- `( )` - parameter list
- `{ }` - function body

This lambda does nothing.

SIMPLE UNNAMED LAMBDA

```
[](int number) { return number % 2; };
```

- `[ ]` - empty capture list
- `(int number)` - takes one parameter - `int`
- `{ return number % 2; }` - lambda body

SIMPLE NAMED LAMBDA

```
auto isOdd = [](int number) { return number % 2; };
```

- `auto` - the only proper type of lambda; it is deduced by the compiler
- `isOdd` - name of the lambda expression
- `[]` - empty capture list
- `(int number)` - takes one parameter - `int`
- `{ return number % 2; }` - lambda body

CALLING A LAMBDA

NAMED LAMBDA

```
auto isOdd = [](int number) { return number % 2; };
auto result = isOdd(101);
```

UNNAMED LAMBDA

```
auto result = [](int number) { return number % 2; }(101);
```

EXAMPLE

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    []() { std::cout << "Hello"; };    // lambda printing Hello (not called)

    std::vector<int> vec {1, 2, 3, 4, 5, 6, 7, 8, 9};
    vec.erase(std::remove_if(vec.begin(),
                             vec.end(),
                             [](int num) { return num % 2; }),
              vec.end());

    auto print = [](int num) { std::cout << num << ' '; };
    std::for_each(vec.begin(), vec.end(), print);

    return 0;
}
```

Output: 2 4 6 8

WHEN TO USE NAMED/UNNAMED LAMBDA?

UNNAMED LAMBDAS

- can be used and called only in one place
- have a local scope
- usually no lifetime issues (when no concurrency)

NAMED LAMBDAS

- reusable
- may be problematic because of lifetime issues

FULL LAMBDA DECLARATION (C++17)

```
[captures](params) specifiers [[attributes]] -> ReturnType { body };
```

FULL LAMBDA DECLARATION (C++20)

```
[captures]<template params>(params) specifiers [[attributes]] -> ReturnType re
```

LAMBDA DESCRIPTION ON CPPREFERENCE.COM

```
[captures](params) specifiers [[attributes]] -> ReturnType { body };
```

```
{ body }
```

Hopefully, explanation is not required 🙃

Every function must have a body. Forward declaration of lambda is not possible.

```
[captures](params) specifiers [[attributes]] -> ReturnType { body };
```

`-> ReturnType`

- Usually, we do not indicate what is a lambda return type, because it is deduced automatically by the compiler. It uses lambda body `{}` to deduce the returned type.
- You can specify a return type of lambda function with arrow notation.

```
[](int rhs, int lhs) -> int { return rhs + lhs; }
```

- From C++14 compiler can easily deduce the returned type, so arrow notation is not popular. It may be used to do implicit conversion.

```
auto isNotNullptr = [](void* ptr) -> bool {
    return ptr;
};
```

- Return type declaration may be needed in case of some template magic.

```
[captures](params) specifiers [[attributes]] -> ReturnType { body };
```

`[[attributes]]`

Barely used. Please refer to Modern C++ training or just check `[[attributes]]` on cppreference.com

```
[captures](params) specifiers [[attributes]] -> ReturnType { body };
```

SPECIFIERS

There are 3 types of lambda specifiers (in C++17):

- `noexcept`
- `constexpr`
- `mutable`

From C++20 `consteval` will be available as well.

`noexcept`

`noexcept` specifier guarantees, that no exception will be thrown from this lambda function.

It is not checked during compilation.

If, in runtime, exception will be thrown then the application will be terminated.

```
auto loggedSwap = [&](auto & a, auto & b) noexcept {
    LOG << "before: a = " << a << ", b = " << b << '\n';
    std::swap(a, b);
    LOG << "after: a = " << a << ", b = " << b << '\n';
};
```

`constexpr`

`constexpr` means that this lambda can be evaluated at compile time and the compiler may used already computed value instead of calling this lambda at runtime.

`constexpr` is implicitly added in C++17 lambdas, wherever possible.

> *If the `constexpr` specifier is omitted within the lambda-declarator, the function call operator (or template) is `constexpr` if it would satisfy the requirements of a constexpr function:*
>
> ```cpp
> auto ID = [](int n) { return n; };
> constexpr int I = ID(3);
> ```
>
> -- *from P0170R1*

`mutable`

`mutable` means, that we can modify const objects captured by the lambda.

`mutable` is associated with capture list, so it will be covered together with it.

```
[captures](params) specifiers [[attributes]] -> ReturnType { body };
```

(params)

Probably it does not require an explanation, but...

- `()` empty parentheses can be skipped

```
auto debugLog = [] { DEBUG << "hello!\n"; };
```

- `()` cannot be skipped when return type is provided or specifiers or attributes are used

```
auto loggedSwap = [&]() noexcept {
    LOG << "before: a = " << a << ", b = " << b << '\n';
    std::swap(a, b);
    LOG << "after: a = " << a << ", b = " << b << '\n';
};
```

FUNNY SNIPPETS

- `[](){};`
  - empty unnamed lambda
- `[]{}();`
  - immediate call of empty unnamed lambda, empty `()` are skipped
- `[](){}();`
  - the same, but `()` is not skipped
- `[]<>(){};`
  - C++20 template lambda
- `(+[](){})();`
  - code::dive 2019 T-shirts snippet, explanation

BREAK

☕

RECAP

WHAT WERE WE TALKING ABOUT BEFORE THE BREAK?

CAPTURE LIST

```
[captures](params) specifiers [[attributes]] -> ReturnType { body };
```

[captures]

Captures variables from the local scope of the lambda.

Things used inside lambda body have another scope. To pass something to lambda you need to use parameters or a capture list.

```
int divide(int number) {
    int divider = 100;
    auto divideBy100 = [divider](int value){
        return value / divider;
    }
    return divideBy100(number);
}
```

WHY THERE ARE 2 WAYS OF PASSING THINGS INSIDE A LAMBDA?

Lambda may be created in one scope and called in another.

Variables from the local scope are captured by a capture list immediately (on lambda creation), whereas parameters may be passed later in a calling scope.

Things from a capture list exist as a lambda internal members.

CAPTURE LIST VALUES

- `[ ]` - capture nothing
- `[a]` - capture variable `a` by copy (value)
- `[&a]` - capture variable `a` by reference
- `[a, &b]` - capture `a` by copy and `b` by reference
- `[=]` - capture everything by copy
- `[&]` - capture everything by reference
- `[&, a]` - capture `a` by copy and everything else by reference
- `[=, &a]` - capture `a` by reference and everything else by copy
- `[this]` - capture `this` pointer **by reference** (lambda can modify current class object)
- `[*this]` - (C++17) capture `this` pointer by copy (creates a copy of current object)

`this` is captured anyway when either `[=]` or `[&]` are used and **in both cases allow to modify** the current object

WHEN USING [&] MAY NOT BE SAFE?

EXAMPLES

```cpp
int a {5};
auto add5 = [=](int x) { return x + a; };
// auto add5 = [](int x) { return x + a; }; // error: a is not defined

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;
```

QUIZ

- `[i, i]`
  - error, `i` repeated
- `[&, &i]`
  - error, by-reference capture when by-reference is the default
- `[this, *this]`
  - error, `this` repeated
- `[&, this]`
  - OK, equivalent to `[&]`
- `[&, this, i]`
  - OK, equivalent to `[&, i]`
- `[=, this]`
  - C++17 - error, since C++20 OK, same as `[=]`

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; };
```

Variable initialized on capture list have `auto` type. It is deduced from the expression.

EXERCISE

`02_asterisks.cpp`

Create a lambda, which prints `*` characters. Each lambda call should produce a string longer by one `*`.

Example:

- `*`
- `**`
- `***`
- etc.

CLOSURE

WHAT IS UNDERNEATH LAMBDA?

Every lambda expression cause the compiler to create 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.

```cpp
auto lambda = [scale, &counter, divider = 100u](int x, int y) {
    return (x + y + counter++) * scale / divider;
};
```

```cpp
struct UnnamedClosureClass { // code generated by the compiler
    UnnamedClosureClass(double _scale, int& _counter)
        : scale(_scale), counter(_counter), divider(100u)
    {}

    inline auto operator()(int x, int y) const {
        return (x + y + counter++) * scale / divider;
    }

private:
    const double scale;      // implicit const
    int& counter;
    const unsigned divider; // implicit const
};

auto lambda = UnnamedClosureClass{scale, counter};
```

`mutable` KEYWORD

`mutable` SPECIFIER

> *`mutable` specifier - permits modification of the class member declared mutable even if the containing object is declared const.*
>
> *-- cppreference.com:*

M&M RULE (MULTITHREADING)

The usual use of `mutable` is in M&M rule.

```
class ThreadsafeCounter {
    mutable std::mutex m; // The "M&M rule": mutable and mutex go together
    int data = 0;

 public:
    int get() const {
        std::lock_guard<std::mutex> lk(m);
        return data;
    }
    void inc() {
        std::lock_guard<std::mutex> lk(m);
        ++data;
    }
};
```

`mutable` IN LAMBDA

Variables captured by value (copy), are implicitly const. They are read-only and you can not modify them.

`mutable` in lambda means, that we can modify objects captured by value.

```
void foo() {
    double number = 0.5;
    [number]() mutable { number += 1.0 }(); // ok
    // [number] { number += 1.0 }();
    // error: assignment of read-only variable ‘number’
    [&number]() { number += 1.0 }(); // ok, capture by ref
}
```

`mutable` keyword allows you to modify const members

EXAMPLE

```cpp
int main() {
    std::vector<int> vec(10);
    std::generate(begin(vec), end(vec), [i{0}]() mutable { return i++; });
    print(vec);

    return 0;
}
```

Output: 0 1 2 3 4 5 6 7 8 9

`std::function` VS POINTER TO FUNCTION

EXERCISE

`03_area.cpp`

Change function `areaLessThan20` into lambda.

Then change it into `areaLessThanX`, which takes `x = 20` on a capture list.

What is the problem?

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

CONVERSION TO POINTER TO FUNCTION #1

Lambda that does not capture anything can be converted into a pointer to function.

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

auto foo(int (*fptr)()) {
    auto result = fptr();
    return result;
}

int main() {
    auto result = foo([] { return 5; });
    std::cout << result;
    return 0;
}
```

CONVERSION TO POINTER TO FUNCTION #2

Lambda that captures can not be converted into a pointer to function.

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

auto foo(int (*fptr)()) {
    auto result = fptr();
    return result;
}

int main() {
    int value = 5;
    // auto result = foo([value] { return value; });
    // error: cannot convert ‘main()::<lambda()>’ to ‘int (*)()’
    // std::cout << result;
    return 0;
}
```

std::function

std::function is a special wrapper class that can hold any type of callable, especially all kind of lambdas.

It is defined in <functional> header.

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

auto foo(std::function<int()> fptr) {
    auto result = fptr();
    return result;
}

int main() {
    int value = 5;
    auto result = foo([value] { return value; });
    std::cout << result;
    return 0;
}
```

FUNCTION SIGNATURES

POINTERS TO FUNCTIONS

- `int (*f)()` - f takes no arguments and returns and int
- `void (*f)(int)` - f takes an int and returns nothing
- `double (*f)(int, string)` - f takes an int and a string and returns double

To get `std::function` template type just remove the pointer name `(*f)`. Name the whole `std::function<>` with the name `f`.

`std::function`

- `std::function<int()> f` - f takes no arguments and returns and int
- `std::function<void(int)> f` - f takes an int and returns nothing
- `std::function<double(int, string)> f` - f takes an int and a string and returns double

`std::function`

> *`std::function` is a general-purpose polymorphic function wrapper. Instances of std::function can store, copy, and invoke any Callable target - functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members.*
>
> *-- from cppreference.com*

Because of this 'polymorphic' feature `std::function` is considered as a heavy stuff. If there is a possibility use pointers to functions instead.

EMPTY `std::function`

`std::function` has a call operator function - `operator()`, that forwards all arguments to the wrapped callable and invokes it.

`std::function` can be empty. Invoking an empty `std::function` results in `std::bad_function_call` exception being thrown.

EXERCISE

`04_invoke.cpp`

Write a function `callAnything()` that can take any function/functor/lambda as a first parameter.

The rest of parameters will be passed into this callable from the first argument.

Function should return a result of a function object call.

```cpp
int main() {
    callAnything(getIndexGenerator);
    callAnything(sum, 5, 6);
    callAnything([]{});
    callAnything([]{ return "Hello!"; });
    callAnything([] { std::cout << "Just testing\n"; });
    callAnything(createVector<int>, std::initializer_list<int>{1, 2, 3});
    return 0;
}
```

Hint #1: Variadic template can handle any number of template parameter.

Hint #2: `std::invoke` may be useful.

BREAK

☕

RECAP

WHAT WERE WE TALKING ABOUT BEFORE THE BREAK?

LIFETIME ISSUES

EXERCISE

`05_dangling.cpp`

Fix the code.

We want a generator to provide consecutive numbers starting from 0.

Correct output: `0123456789`

DANGLING REFERENCE

The main problem with lambdas is a dangling reference.

Lambda is created in a place of it's definition - on stack. Some variables are available during it's creation.

This lambda does not have to be called immediately or in a current scope. It may be passed somewhere else. Some local variables captured by lambda by reference may become unavailable. This causes UB (Undefined Behavior).

EXERCISE

`06_synchronization.cpp`

Fix `generateContainer()` function.

We want to have each value from 1 to 10 to be printed on the screen only once.

Every thread should put only one value to the shared vector.

The order does not matter.

Possible outputs:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
2, 4, 6, 8, 10, 1, 3, 5, 7, 9
```

```
1, 3, 2, 4, 5, 6, 8, 10, 9, 7
```

DANGLING REFERENCE WITH `std::function`

Care should be taken when `std::function` whose ***result type is a reference*** *is initialized from a lambda expression [...]. Due to the way auto deduction works, such lambda expression will always return an prvalue. Hence, the resulting reference will usually bind to a temporary whose lifetime ends when `std::function::operator()` returns.*

```
std::function<const int&()> F([]{ return 42; });
int x = F(); // Undefined behavior:
// the result of F() is a dangling reference
```

-- *cppreference.com*

`constexpr` LAMBDA

IMPLICIT `constexpr`

From C++17 all lambda functions are by default implicitly marked as `constexpr`, if possible.

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

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

EXPLICIT `constexpr`

`constexpr` keyword can also be used explicitly.

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

ADVANCED EXAMPLE

`static_assert` is evaluated at compile time. Only `constexpr` values can be used in it.

```
constexpr auto add = [](int n, int m) {
    auto L = [=] { return n; };
    auto R = [=] { return m; };
    return [=] { return L() + R(); };
};
static_assert(add(3, 4)() == 7);
```

THE 'LAMBADA' ANTIPATTERN

Above code with lambdas inside lambdas is difficult to understand. It takes some time to deduce the way it works. If it is not necessary - avoid it :)

Lifetime issues may be especially arduous to discover and debug.

LUNCH BREAK

RECAP

WHAT WERE WE TALKING ABOUT BEFORE THE BREAK?

GENERIC LAMBDA

GENERIC LAMBDAS

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

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

```
int main() {
    auto multiplyByFactor = [factor = 10](auto number) {
        return number * factor;
    };
    std::cout << multiplyByFactor(20) << '\n';
    std::cout << multiplyByFactor(1.23) << '\n';

    return 0;
}
```

Output:

```
200
12.3
```

GENERIC LAMBDAS CLOSURE CLASS

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 a compiler generates a code equivalent to a closure class given below:

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

```
struct UnnamedClosureClass {
    template <typename T1, typename T2>
    auto operator()(T1 x, T2 y) const {
        return x + y;
    }
};
auto lambda = UnnamedClosureClass{};
```

TEMPLATES VS GENERIC LAMBDAS

Templates are the least intuitive feature of C++, however, very useful one.

Generic lambdas can replace some template function.

```cpp
#include <iostream>
#include <memory>

template <typename T>
std::unique_ptr<T> wrapWithUniquePtr(T value) {
    return std::make_unique<T>(value);
}

int main() {
    auto ptr1 = wrapWithUniquePtr(4);
    auto ptr2 = wrapWithUniquePtr(5.3);

    return 0;
}
```

```cpp
auto wrapWithUniquePtr = [](auto value) {
    return std::make_unique<decltype(value)>(value);
};
```

EXERCISE (OR HOMEWORK)

`07_sfinae.cpp`

Write a function `insert()` that allows objects of class derived from Circle to be inserted into collection.

Objects of other class should be not allowed.

Usage:

```
insert(Circle{1.0}, circles);
insert(Ellipse{1.1}, circles);       // ok, derives from Circle
// insert(double{1.1}, circles);    // not allowed
```

You may use SFINAE (`std::enable_if`) or `constexpr if`.

SFINAE SOLUTION

```cpp
template <typename T,
          typename Collection,
          typename = std::enable_if_t<std::is_base_of_v<Circle, T>>
>
void insert(T item, Collection& collection) {
    collection.emplace_back(make_shared<T>(item));
}
```

if constexpr SOLUTION

```
auto insert = [](auto item, auto& collection) {
    if constexpr (std::is_base_of_v<Circle, decltype(item)>) {
        collection.emplace_back(make_shared<decltype(item)>item);
    }
};
```

WHAT IS A DIFFERENCE BETWEEN THOSE TWO SOLUTIONS?

- SFINAE does not compile (or does not generate an additional overload) for improper types
- `constexpr if` does nothing
- Lambda is an object with some lifetime, it must be created in a proper place (or must be passed)
- Function is not an object (in C++), so it can be called anywhere. Proper include is needed.

RECAP

WRITE IN A CHAT WINDOW 5 THINGS THAT YOU REMEMBER BEST FROM TODAY'S SESSION.

- STL algorithms customization (30")
- anatomy of lambda (20")
- capture list and mutable lambdas (30")
- std::function vs pointer to function (60")
- lifetime issues (40")
- constexpr lambdas (10")
- generic lambdas (10")
- templates vs generic lambdas (40")

PRE-TEST

ANSWERS

1. 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; }`

2. WHICH CAPTURE LISTS ARE CORRECTLY FORMED?

Assume that all variables exist.

1. `[=, this]`
2. `[&, this]`
3. `[this, *this]`
4. `[&, &a]`
5. `[a, &]`
6. `[&, a, &b]`
7. `[=, a, &b]`
8. `[=, &a]`
9. `[=, *a]`

POST-TEST

The link to post-test will be sent to you in a next week.

It's better to forget some of the content and refresh your knowledge later.

It enhances knowledge retention :)

HOMEWORK

Write a `schedule()` function. It should be able to run the code asynchronously and provide a feedback when this code is finished.

It should take 2 arguments:

- any callable that will be run in a new thread - asynchronously
- a callback function, which will be called when the first function will be completed

Send me a link to the repository with your solution.

FEEDBACK

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

THANK YOU 🙂