C++

Fall 2022

compscicenter.ru

Башарин Егор

eaniconer@gmail.com
https://t.me/egorbasharin

Лекция XI

Lambda expressions. STL functional objects

Section 1

Motivation

Motivation example

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

struct EvenPrinter {
    void operator()(int i) const {
        if (i % 2 == 0) std::cout << i;
    }
};

int main() {
    std::vector<int> v {1,2,3,4,5,6};
    std::for_each(v.begin(), v.end(), EvenPrinter{});
}
```

Motivation example

Слишком широкая область видимости, но можно использовать локальный класс:

```
int main() {
    struct EvenPrinter {
        void operator()(int i) const {
            if (i % 2 == 0) std::cout << i;
        }
    };

    std::vector<int> v {1,2,3,4,5,6};
    std::for_each(v.begin(), v.end(), EvenPrinter{});
}
```

Motivation example

Слишком много не нужного кода, хотелось бы проще — оставить только параметры и тело, например так:

```
int main() {
    std::vector<int> v {1,2,3,4,5,6};
    std::for_each(v.begin(), v.end(),
        (int i) { std::cout << i; });
}
```

Section 2

Lambda expression type, parameters, return type

The simplest lambda

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

- Тип лямбда выражения скрыт (генерируется компилятором), поэтому для сохранения результата выражения пользуемся `auto`
- Считаем результат функциональным объектом и пользуемся `operator()` для вызова

The simplest lambda

```
int main() {
    auto lambda = [](){};
    lambda();
}
```

Lambda anatomy: cppinsights

Simple lambda

```cpp
int main() {
    auto lambda = [](int param1, double param2 = 3) -> i
        return param1 + param2;
    };

    int(*ptr)(int, double) = lambda;

    std::cout << lambda(1);
    std::cout << ptr(1, 2.0 /*param2 required*/);
}
```

cppinsights

Lambda syntax

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

```
(*) captures рассмотрим позже
```

Section 3

Generic lambda

Generic lambda example

```
int main() {
    auto f = [](auto x, auto& y, auto&& z) {
        return x + y + z;
    };
    int x = 1;
    f(1, x, 3);
}
```

cppinsights

Perfect forwarding problem

```cpp
#include <iostream>
#include <utility>

struct T {};
int p(const T&) { return 1; }
int p(T&&) { return 2; }

int main() {
    auto f = [](auto&& t){ return p(t); };

    T t;
    assert(f(t) == 1);
    assert(f(T{}) == 2); // ?
    assert(f(std::move(t)) == 2); // ?
}
```

Perfect forwarding

```
struct T {};
int p(const T&) { return 1; }
int p(T&&) { return 2; }

int main() {
    auto f = [](auto&& t){
        return p(std::forward<decltype(t)>(t));
    };

    T t;
    assert(f(t) == 1);
    assert(f(T{}) == 2);
    assert(f(std::move(t)) == 2);
}
```

Section 4

Capturing

Variables in lambda’s body

```cpp
double global = 10; // OK
static float global_static = 20; // OK

int f() { return 42; }

int main() {
    static int local_static = 1; // OK
    const int integral_const_1 = 2; // OK (if not odr-used)

    const float float_const = 1.0f; // NO
    const int integral_const_2 = f(); // NO

    auto lambda = [] {
        std::cout << global
        << " " << global_static
        << " " << local_static
        << " " << integral_const_1
        << " " << float_const
        << " " << integral_const_2;
    };
    lambda();
}
```

Capture

```cpp
#include <iostream>

int main() {

    int x = 1;
    int y = 2;

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

    std::cout << lambda();
}
```

Capture by value (copy)

```
int main() {
    int x = 1;
    auto f = [x](){ return 2*x; }; // explicit capture
    auto g = [=](){ return 2*x; }; // implicit capture
    auto h = [i = x](){ return 2*i; }; // with initializ
}
```

- Вывод типов при использовании инициализатора отличается
- В теле лямбды нельзя изменять значения "захваченных" переменных

cppinsights

Capture by value + `mutable`

```cpp
#include <cassert>

int main() {
    int x = 1;

    auto f = [x]() mutable { x = 20; return x; };
    f();

    assert(x == 1);
}
```

Capture by reference

```
int main() {
     int x = 1;

     auto f = [&x]() { return 2*x; }; // explicit captur
     auto g = [&]() { return 2*x; };  // implicit captur
     auto h = [&i = x]() { return 2*i; }; // with initia
}
```

- Типы выводятся также как для шаблонов
- Изменение переменной в теле лямбды влиет на значение объекта, ссылка на который, была захвачена

cppinsights

Capture by const-reference

```
#include <utility>

int main() {
     int x = 1;
     auto f = [&x = std::as_const(x)]() { return 2*x; };
}
```

Combining captures

```
int main() {
    int x = 1;
    int y = 2;

    auto f = [x, &y] { return x * y; };
    auto g = [=, &y] { return x * y; };
    auto h = [&, x] { return x * y; };
}
```

Capture members

```
#include <iostream>

struct S {
    void test() {
        auto f = [](){
            std::cout << data_member; // error
        };
        f();
    }
    int data_member = 42;
};
```

Capture members

```
#include <iostream>

struct S {
    void test() {
        auto f = [](){
            std::cout << data_member; // error
        };
        f();
    }
    int data_member = 42;
};
```

Ways:

```
[data_member = data_member]
[&data_member = data_member]
[=]
[&]
[this]
[*this]
```

Capture members (test)

```cpp
#include <iostream>

struct S {
    auto lambda() {
        return [ ??? ]() { return data_member; };
    }
    ~S() { data_member = 0; }
    int data_member = 10;
};

auto make_lambda() {
    S s;
    return s.lambda();
}

int main() {
    auto f = make_lambda();
    std::cout << f();
}
```

Какой из перечисленных ранее способов захвата приведет к UB из-за висячей ссылки (dangling reference)?

Capturing movable-only objects

```
#include <memory>

int main() {
    std::unique_ptr<int> p = std::make_unique<int>(10);
    auto f = [p]() { return *p; }; // Error
}
```

Capturing movable-only objects

Без передачи владения:

```
auto f = [&p]() { return *p; };
```

С передачей владения:

```
auto f = [p = std::move(p)]() { return *p; };
```

Section 5

Lambda tricks

IILE idiom

Immediately Invoked Lambda Expression

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

int main() {
    const std::vector<int> vs = []{
        std::vector<int> vs(10);
        std::iota(vs.begin(), vs.end(), 10);
        return vs;
    }();

    std::copy(
        vs.begin(), vs.end(),
        std::ostream_iterator<int>(std::cout, " "));
}
```

Обычно используется для нетривиальной инициализации

Lifting

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

void process(int) {}
void process(double) {}

int main() {
    {
        std::vector<int> ints {1,2,3};
        std::for_each(ints.begin(), ints.end(), process);
    }

    {
        std::vector<double> doubles{1.0, 2.0};
        std::for_each(doubles.begin(), doubles.end(), process);
    }
}
```

for_each documentation

Lifting

Проблема выбора нужной перегрузки функции process

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

void process(int) {}
void process(double) {}

int main() {
    {
        std::vector<int> ints {1,2,3};
        std::for_each(ints.begin(), ints.end(), process);
    }

    {
        std::vector<double> doubles{1.0, 2.0};
        std::for_each(doubles.begin(), doubles.end(), process);
    }
}
```

Lifting

Fix using lambda

```
#include <algorithm>
#include <vector>

void process(int) { }
void process(double) { }

int main() {
    {
        std::vector<int> ints {1,2,3};
        std::for_each(ints.begin(), ints.end(),
                      [](int x) { process(x); });
    }

    {
        std::vector<double> doubles{1.0, 2.0};
        std::for_each(doubles.begin(), doubles.end(),
                      [](double x) { process(x); });
    }
}
```

Lifting

Add macros to generalize

```
#include <algorithm>
#include <vector>

void process(int) { }
void process(double) { }

#define LIFT(fun) [](auto x) { (fun)(x); }

int main() {
    {
        std::vector<int> ints {1,2,3};
        std::for_each(ints.begin(), ints.end(), LIFT(process));
    }

    {
        std::vector<double> doubles{1.0, 2.0};
        std::for_each(doubles.begin(), doubles.end(), LIFT(process));
    }
}
```

Section 6

STL functional objects library

#include <functional>

functional objects library

- `function`
- `bind`
- comparisons: `less`, `greater`, etc...
- ...

std::function

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

int square(int x) { return x * x; }

void run(const std::function<int(int)>& func, int arg) {
    std::cout << func(arg) << " ";
}

struct FuncObj {
    int operator()(int i) const { return i * i; }
};

int main() {
    run(square, 10); // free-function

    run([](int x) { return x; }, 1); // lambda without capture

    run([i = 10](int x) { return i + x; }, 2); // lambda with capture

    run(FuncObj{}, 11); // functor
}
```

std::function

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

struct T {
    int method(int i) const { return i; }
};

int main() {
    T t;

    std::function<int(const T*, int)> f = &T::method;
    std::cout << f(&t, 1) << " ";

    std::function<int(const T&, int)> g; // may be empty
    if (!g) { // check emptiness
        g = &T::method;
        std::cout << g(t, 2) << " ";
    }
}
```