

Lecture 25

Lambdas

Почему лямбда это не функция, а класс? Потому что у функция требует памяти под указатель, тогда как лямбда - нет.

```
1 // lambda
2 [capture_list] (args_list) -> return_type
3 // "(...)" and "-> return_type" can be omitted
```

Обсуждение

- λ -выражения это не функции

```
auto t = [z](auto x, auto y) { return x < y * z; };
```

- Это скорее классы с перегруженным operator()

```
struct __closure_type_for_t {
    int __k;
    auto operator()(auto x, auto y) const {
        return x < y * __k;
    }
} t{z};
```

9

<https://godbolt.org/z/Woe4hxs55>

`__k` - захват, захваченное состояние.

В лямбдах захват копируется.

По умолчанию захват имеет const qualifier. Если мы хотим менять состояние - явно пишем `mutable`.

Убираем const

- Если мы хотим изменять захваченный по значению контекст мы должны сделать нашу лямбду в явном виде mutable
- ```
auto t = [z](auto x, auto y) mutable { z += 1; return x < y * z; };
```
- Обратите внимание, что z изменяется в пределах замыкания.
- ```
auto s = t;
```
- ```
t(1, 2); // z изменилось внутри t, но не внутри s
```
- Замыкания по умолчанию копируемые (если не захвачено ничего со стёртым сорусор).
  - Глобальные и статические переменные захватывать не надо, они доступны и так.

10

Глобальные и статические переменные и так видны в лямбде.

```
1 [a](){} // захват по значению
2 [&a](){} // захват по ссылке (сама ссылка immutable, но то, на что она
3 ссылается - да.)
```

## Виды захвата

- Захват по значению (по ссылке)
- ```
auto fval = [a, b](int x) { return a + b * x; };
auto fvalm = [a, b](int x) mutable { a += b * x; return a; };
auto fref = [&a, &b](int x) { a += b * x; return a; };
```
- Захват по ссылке всегда mutable и отслеживает состояние переменной.
- ```
a = 42;
fval(x); // тот же
fref(x); // использует новое a
```
- Разумеется можно смешивать: [&a, b, &c, d]

11

## Виды захвата

- Захват всего контекста по значению (по ссылке)

```
auto faval = [=](int x) { return a + b*x; };
auto faref = [&](int x) { a += b*x; return a; };
```

- Захват всего по значению и частично по ссылке и наоборот

```
auto favalb = [=, &b](int x) { return a + b*x; };
auto farefa = [&, a](int x) { b += a*x; return b; };
```

- Захват с переименованием

```
auto freval = [la = a](int x) { return la + x; };
auto freref = [&la = a](int x) { la += x; return la; };
```

12

<https://godbolt.org/z/hb5E4b5a6>

Почему адреса нельзя захватить по ссылке - потому что lval нельзя присвоить rval.

Захват с перемещением

## Виды захвата

- Переименование позволяет захватить с перемещением.

```
std::ostream_iterator<int> os{std::cout, " "};
std::vector v = {1, 2, 3};

auto out = [w = std::move(v), os] {
 std::copy(w.begin(), w.end(), os);
};
```

- Теперь вектор передан в состояние замыкания.
- Передача осуществляется в конструкторе замыкания, т.е. в момент создания функции, а не в момент её вызова.

13

<https://godbolt.org/z/gYd3fEKe8>

Захват в теле класса

## Захват в теле класса

```
struct Foo {
 int x;
 void func() {
 [x] mutable { x += 3; } (); // FAIL
 [&x] { x += 3; } (); // FAIL

 [=] { x_ += 3; } (); // OK
 [&] { x_ += 3; } (); // OK

 [this] { x_ += 3; } (); // OK
 }
};
```

- Это работает, поскольку полный захват захватывает `this`

14

<https://godbolt.org/z/ar46ax95P>

```
4 struct Foo {
5 int x = 5;
6 void xplus() {
7 #if 0
8 [x] mutable { x += 3; } (); // FAIL
9 [&x] { x += 3; } (); // FAIL
10 #endif
11 [=] { x += 3; } (); // not exactly ok (deprecated in C++20)
12 [&] { x += 3; } (); // OK
13 [this] { x += 3; } (); // OK
14 }
15 };
16
17 int main() {
18 Foo f;
19 f.xplus();
20 std::cout << f.x << std::endl;
```

Задача на closure

Важно помнить, что лямбда - это скорее класс с перегруженным оператором `()`.

## Задача: локальный контекст

```
auto factory (int parameter) {
 static int a = 0;
 return [=] (int argument) {
 static int b = 0;
 a += parameter; b += argument;
 return a + b;
 }
}

auto func1 = factory(1); auto func2 = factory(2);

cout << func1(20) << " " << func1(30) << " "
 << func2(20) << " " << func2(30) << endl;
```

15

<https://godbolt.org/z/xr5Ma8cyY>

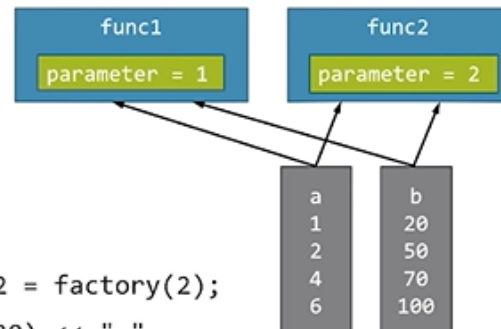
У нас внутри функции одно определение класса, в котором один перегруженный оператор круглые скобки.

## Решение: локальный контекст

```
auto factory (int parameter) {
 static int a = 0;
 return [=] (int argument) {
 static int b = 0;
 a += parameter; b += argument;
 return a + b;
 };
}

auto func1 = factory(1); auto func2 = factory(2);

cout << func1(20) << " " << func1(30) << " "
 << func2(20) << " " << func2(30) << endl;
```



16

Проблема поля класса, инициализированного лямбдой:

## Единая типизация замыканий

```
auto t = [&x, &y] { return x + y; };
```

- Тут не вполне ясно что такое `auto`.
- Мало того, так нельзя делать в теле класса.

```
struct Foo {
 auto t = [&x, &y] { return x + y; }; // ошибка
};
```

- Мы понимаем, что это `closure type` но не можем записать его явно.
- В этом случае мы можем частично стереть тип.

17

## Единая типизация замыканий

```
auto t = [&x, &y] { return x + y; }; // closure
```

```
function<int()> f = [&x, &y] { return x + y; }; // function
```

- `std::function<сигнатура>` это единый тип к которому **приводятся** все замыкания с данной сигнатурой.

- Ключевое слово "приводятся".

```
t = [] { return 42; }; // FAIL
```

```
f = [] { return 42; }; // ok
```

- Тип функции теряет информацию о захвате контекста. Значение имеет только сигнатура.



18

Параметр `std::function` - тип возвращаемого значения и аргументы функции. Она позволяет обобщать лямбды по `catch_lists`

*Наблюдается общность с таблицей виртуальных функций*

## Снова захват в теле класса

```
using VVTy = std::function<void(void)>;
```

```
struct Foo {
 int x;
```

```
 VVTy xplus1 = [&] { x += 3; }; // OK, but hmmm...
```

```
 VVTy xplus2 = [this] { x += 3; }; // OK
```

- Это вдвойне интересно, так как такие лямбды-члены ведут себя как методы.

```
struct Foo f;
```

```
f.xplus1(); // OK
```

19

<https://godbolt.org/z/YqWGTaze>

## Case study: finally

- Допустим у нас есть очень простой класс:

```
struct Finally {
 std::function<void()> action_;
 explicit Finally(std::function<void()> action):
 action_(std::move(action)){}
 ~Finally() { action_(); }
};
```

- Теперь мы можем вот такие фокусы

```
FILE *f = fopen("myfile.dat", "r"); assert(f);
Finally close_f([&f]{ fclose(f); });
```

21

<https://www.youtube.com/watch?v=eG5suWcHI8M>

Классная техника. Но в ней есть один недостаток - `std::function` инициализирует данные на куче - это небесплатно относительно следующего...

Мы могли бы занести `action` в шаблонн, но тогда при специализации указание лямбды породило бы класс

## Алгоритмы

### Алгоритмы

- Алгоритм стандартной библиотеки это **функция**, выполняющая действие над **интервалами**, заданными с помощью итераторов.

```
template<class InputIt, class OutputIt>
OutputIt copy(InputIt fst, InputIt last, OutputIt res);
```

- Имя алгоритма может иметь суффиксы и префиксы
  - `if` (например `copy_if`) – выполнить действие при выполнении предиката
  - `n` (например `copy_n`) – выполнить действие ограниченное количество раз
  - `copy` (например `reverse_copy`) – разместить результат в новом контейнере
  - `stable` (например `stable_partition`) – алгоритм работает стабильно
- Вопрос должен ли `copy_n` существовать в языке – дискуссионный.

25