5.18 Лямбда-функции. Объекты анонимного типа, сгенерированные из лямбда-функций (замыкания), их внутреннее устройство. Правила генерации компилятором полей и методов этих объектов. Особенности копирования и присваивания этих объектов. Особенности захвата полей класса и указателя this в лямбда-функции. Опасность захвата по умолчанию.

Выражение стоящее после [.] называется **замыканием** (closure). Оно является rvalue

Синтаксис

```
std::vector<int> v = {1,6,4,6,3,6};
std::sort(v.begin(), v.end(), [](int x, int y) {
    return std::abs(x - 5) < std::abs(y - 5);
});
```

Можно объект проинициализировать лямбда-функцией, тип объекта - ожидаемо auto.

Тип, который имеет лямбда-функция, генерирует компилятор. Он превращает код в некоторое внутреннее представление и дает имя, которое бы не пересекалось с другими именами в программе.

Давайте попробуем узнать размер объекта f

```
auto f = [](int x, int y) {
    return x < y;
};

//g(f);

std::cout << typeid(f).name() << '\n';

std::cout << sizeof(f) << '\n';
```

Программа выдаст 1 байт. Это происходит потому, что у этого объекта нет никаких полей, а все, что есть - один метод, функция f. Можно считать, что все, что у нас есть - это один пустой функциональный класс, сгенерированный компилятором, в котором определен оператор ()

Что будет, если теперь захватить то-то в лямбда-функцию?

```
int a = 1;

auto f = [a](int x, int y) {
    return x + a < y;
};
```

Теперь размер f увеличился до 4 байт, потому что мы захватили объект. То есть в сгенерированном функциональном классе появилось поле int

Что на самом деле происходит, когда мы что-то захватываем в лямбда-функцию?
Компилятор генерирует под каждую функцию специальный класс, тип которого олицетворяет эту лямбда-функцию, в котором он автоматически определяет оператор круглые скобочки, исходя из параметров, переданных в лямбда-функцию. Все захваченные переменные компилятор делает полями созданного функционального класса, и созданные поля инициализирует тем, то мы захватили

Соответственно, если мы захватим по ссылке

```
int a = 1;

auto f = [&a](int x, int y) {
    return x + a < y;
};
```

то теперь размер f будет 8 байт, потому что теперь полем является ссылка на int, а ссылка на int инициализируется a при создании лямбда-функции

Отсюда понятно, почему поля, которые мы захватили, как копии по значению, неизменны. Это происходит потому что оператор () из лямбда-функций по стандарту константный.

Понятно, почему он так должен делать. Потому что всякая нормальная сортировка имеет право считать компаратор константным. Соотвественно, если компилятор создал поля в этом классе, то он считает их неизменными из-за того, то у нас константный метод.

А вот со ссылками так не работат, потому что const в методе не распространяется на ссылки. Если у нас поле int&, то даже если метод константный, поле все равно может меняться из этого метода, потому что const у метода означает навешивание const справа на тип, а не слева

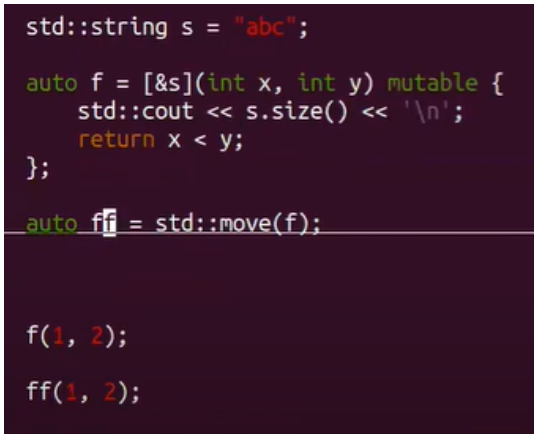
Кроме оператора () компилятор генерирует для функционального класса конструктор копирования (это логично, так как иначе бы нельзя было передавать нашу функцию по значению куда-либо) и мув-конструктор. Первый просто копирует поля, а второе мувает все поля

```
auto f = [a](int x, int y) mutable {
    ++a;
    std::cout << a << '\n';
    return x + a < y;
};

auto ff = f;

f(1, 2);

ff(1, 2);
```

Здесь в обоих случаях выведется 2, изменение a в одной функции не повлияло на a в другой. Если бы a принималось по ссылке, то ответ был бы 2 и 3

```
std::string s = "abc";

auto f = [s](int x, int y) mutable {
    std::cout << s.size() << '\n';
    return x < y;
};

auto ff = std::move(f);

f(1, 2);

ff(1, 2);
```

А здесь мы мувнули строку из f, в результате чего она стала пустой. Выведется 0 и 3

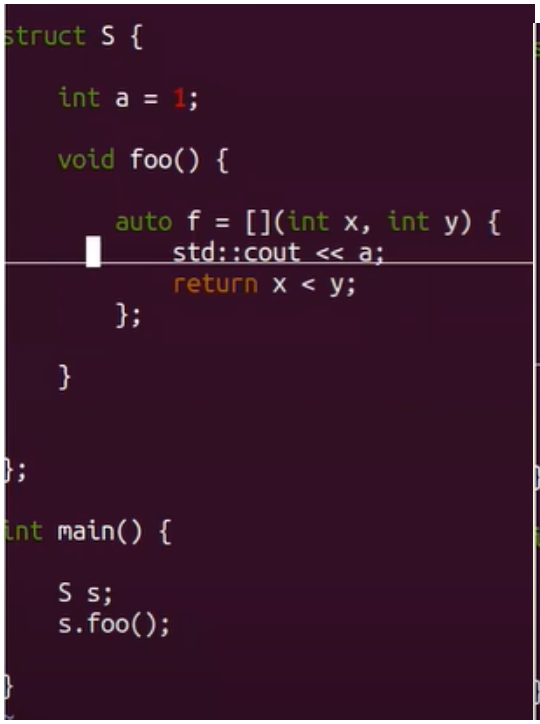


Выведется 3 и 3, поскольку мув ссылки не делает ничего

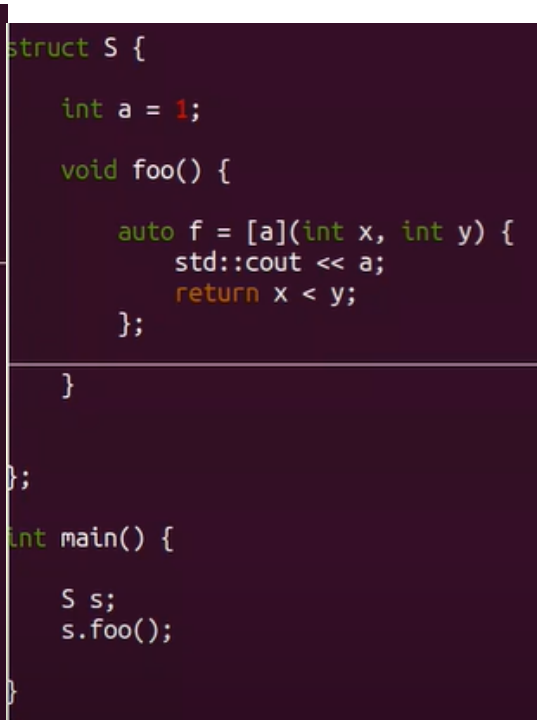
Компилятор не генерирует оператор присваивания, если есть захват! Но начиная с C++20 он может генерировать оператор присваивания, если захвата нет

Начиная с C++20 компилятор генерирует конструктор по умолчанию, до этого выдавал CE

Особенности захвата полей класса и указателя this в лямбда-функции



Не сработает

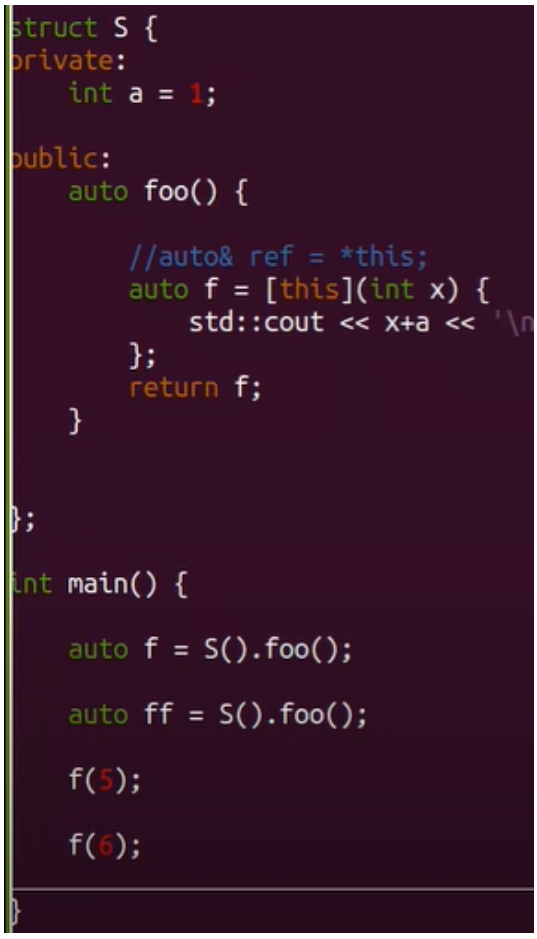


Не сработает

Захватывать можно только локальные переменные, но не поля класса

Но зато можно сделать вот так

В лямбду мы захватываем указатель на текущий объект. Теперь мы можем образаться к полям this из функции



Вот так будет UB, так как функция пережила тот объект, который она захватила

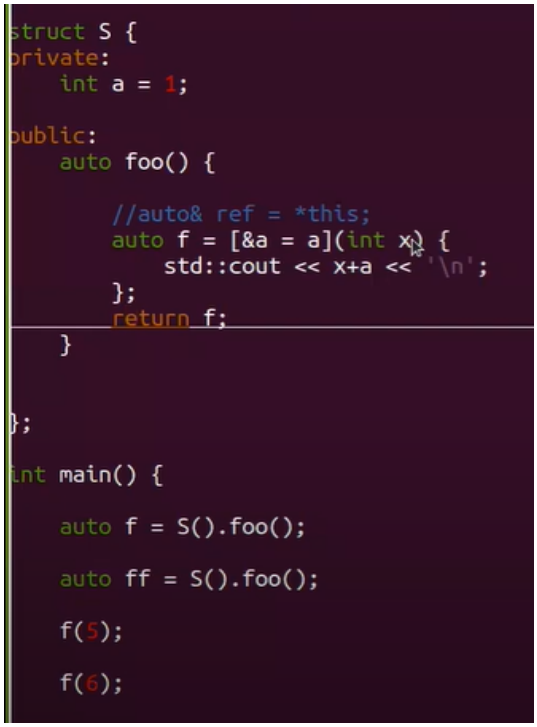
Capture with initialization

Начиная с C++14 появилась возможноть инициализации при захвате

Для компилятора это значит, что мы просим его завести в генерируемом классе поле b, но проинициализировать его посредством a, которое берется из текущей области видимости, включая поля класса

Самое прекрасное, что от нас никто не требует, чтобы поле и то, чем мы его инициализируем, назывались по-разному. То есть такая штука тоже сработает, и это будет не то же самое, что просто захватить a, сейчас не будет никакого UB и все нормально скомпилируется

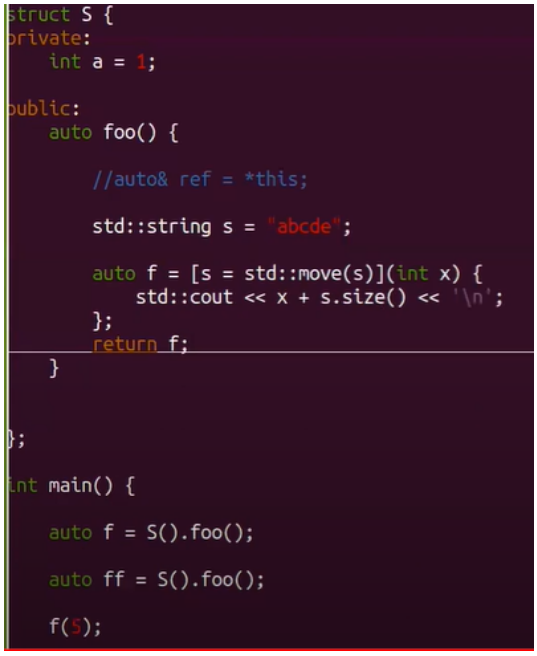
Вот так тоже можно сделать, но мы получим UB, потому что то, чем мы проинициализировали ссылку, умрет раньше, чем функция. И во втором случае также будет UB, по той же причине



```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;
        auto f = [&a = a](int x) {
            std::cout << x+a << '\n';
        };
        return f;
    }


};

int main() {

    auto f = S().foo();

    auto ff = S().foo();

    f(5);

    f(6);
```

```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;
        auto f = [a = &a](int x) {
            std::cout << x + *a << '\n';
        };
        return f;
    }


};

int main() {

    auto f = S().foo();

    auto ff = S().foo();

    f(5);

    f(6);
```

Еще одна важная вещь, которая появилась - возможность захватывать по rvalue ссылке. Делается это с помощью захвата с инициализацией и std::move(1).
Вот мы умеем захватывать по значению, по ссылке, мувать... а можно по константной сылке? Да!(2)

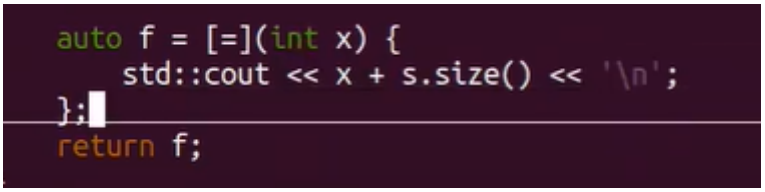


```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;

        std::string s = "abcde";

        auto f = [s = std::move(s)](int x) {
            std::cout << x + s.size() << '\n';
        };
        return f;
    }


};

int main() {

    auto f = S().foo();

    auto ff = S().foo();

    f(5);
```

```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;

        std::string s = "abcde";

        auto f = [&s = std::as_const(s)](int x) {
            std::cout << x + s.size() << '\n';
        };
        return f;
    }


};
```

(1) (2)

Есть синтаксис захвата всех переменных сразу. Можно захватить все локальные переменные по значению или по ссылке. **Но так не надо делать!**. Потому что так или иначе мы забываем, что захватили. И тем самым мы получаем доступ к к переменным, которые на самом деле, возможно, уже умерли



```
auto f = [=](int x) {
    std::cout << x + s.size() << '\n';
};
return f;
```

Так не надо

Приведем пример, почему это плохо

```
struct S {
private:
    int a = 1;

public:
    auto foo() {

        //auto& ref = *this;

        std::string s = "abcde";

        auto f = [=](int x) {
            std::cout << x + a << '\n';
        };
        return f;
    }


};

int main() {

    auto f = S().foo();

    auto ff = S().foo();
```

Вот тут мы как будето захватили все имена по значению, но на деле мы захватили this, то есть захватили еще и a, которое умерло раньше, чем наша функция, и в итоге получилось UB
Еще хуже захват по ссылке. Будет такое же UB

```
auto f = [&](int x) {
    std::cout << x + a << '\n';
};
```

Так не надо

Что можно сделать еще... Сказать "захвати все по ссылке, кроме s"

```
auto f = [&, s](int x) {
    std::cout << x + a << '\n';
};
```

"захвати все по значению, кроме s"

```
auto f = [=, &](int x) {
    std::cout << x + a << '\n';
};
```