

4.31. Лямбда-функции и лямбда-выражения, их синтаксис. Пример использования в стандартных алгоритмах, пример использования в качестве компаратора для `map`. Списки захвата в лямбда-выражениях, их синтаксис, пример использования. Синтаксис захвата по ссылке и по значению.

С C++11 появились лямбда-функции. Они помогают описывать функции прямо внутри выражения, которое их вызывает.

До:

```
1 struct MyCompare {
2     bool operator()(int x, int y) const {
3         return std::abs(x - 5) < std::abs(y - 5);
4     }
5 };
6
7 std::vector<int> v{1, 5, 4, 7};
8 std::sort(v.begin(), v.end(), MyCompare());
```

Основное неудобство - приходится объявлять функцию/компаратор снаружи области видимости - нарушение инкапсуляции.

После, синтаксис:

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

Объявление `[]` - **closure expression**, дальше параметры, дальше тело функции. Можно объект проинициализировать лямбда-функцией, тип объекта - ожидаемо `auto`.

Можно возвращать функции из других функций (пишем лямбда функцию после `return`), в примере ниже написан компаратор, который можно передавать в качестве параметра сортировки)

```
1 auto getCompare() {
2     return [](int x, int y) {
3         return std::abs(x - 5) < std::abs(y - 5);
4     }
5 }
6
7 [](int x) {
8     std::cout << x << "\n"; //declaration, rvalue
9 };
10
11 [](int x) {
12     std::cout << x << "\n"; //called
13 }(5);
```

Тип возвращаемого значения определяется по правилам вывода типов, он установлен по умолчанию. Если так получилось, что компилятор сам не справляется (например две ветки условий, и в каждой возвращаемое значение разное) или мы хотим кастомный `type_deduction`, то можно в явном виде прописать, какого типа вывод мы ожидаем

```
1 [](int x) -> bool {
2     std::cout << x << "\n"; //declaration, rvalue
3 };
```

Capture lists

```
1 int a = 1;
2
3 [](int x) {
4     std::cout << x + a << "\n";
5 }(5);
```

Будет СЕ потому что а не захвачен внутри лямбда-функции, локальные объекты не попадают в область видимости, в отличие от глобальных или конкретных namespace.

Надо было написать так:

```
1 int a = 1;
2
3 [a](int x) {
4     std::cout << x + a << "\n";
5 }(5);
```

Но так нельзя будет менять переменную а - ее тип const int. Чтобы можно было менять надо написать так: [a](int x) mutable {...};

```
1 [a](int x) {
2     std::cout << x + a << "\n";
3     ++a;
4 }(5);
```

Однако здесь тип а - int& !

Лямбда-функция для map:

```
1 auto f = [](int x, int y){...};
2 std::map<int, int, decltype(f)> m;
```