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:

```
struct MyCompare {
    bool operator()(int x, int y) const {
        return std::abs(x - 5) < std::abs(y - 5);
    }
};

std::vector<int> v{1, 5, 4, 7};
std::sort(v.begin(), v.end(), MyCompare());
```

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

После, синтаксис:

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

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)

```
    auto getCompare() {
        return [](int x, int y) {
            return std::abs(x - 5) < std::abs(y - 5);
        }
    }

    [](int x) {
        std::cout << x << "\n"; //declaration, rvalue
    };

    [](int x) {
        std::cout << x << "\n"; //called
    }(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 разниое) или мы хотим кастомный type_deduction, то можно в явном виде прописать, какого типа вывод мы ожидаем

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

Capture lists

```
int a = 1;

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

Будет CE потому что a не захвачен внутрь лямбда-функции, локальные объекты не попадают в область видимости, в отличие от глобальных или конкретных namespace.

Надо было написать так:

```
int a = 1;

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

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

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

Однако здесь тип a - int& !

Лямбда-функция для map:

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