

- Функтор (функциональная сущность) это просто объект, у которого определен оператор **operator()** Нахуя такие нужны? Как минимум для инкапсуляции связанных с ними методов
- Указатели на функцию желательно наглядно типизировать передавая в другие функции, а не писать `auto` в аргументах

Возвращаемый тип функции

Название переменной, содержащей указатель

Список типов аргументов.

Note: `const`-квалификатор здесь не имеет никакого эффекта (так же, как в случае деклараций функций)

```
double function (int a, float b) { return a > 3 && b < 1.5f ? 3.14 : 0.44; }

int main () {
    double (*ptr) (int, float) = &function;
    double res = ptr(2, 2.f);
    std::cout << res << std::endl;
}
```

Note: этот амперсанд можно опустить, однако я привык его писать, так как это явно показывает, что объект является указателем и содержит какой-то адрес.

- Можно создавать также указатели на методы классов и вытаскивать их из определённой сущности класса. Зачем? Кто бы знал...

```
int main () {
    void (vector2d::*scale_ptr) (double) = &vector2d::scale;

    /// Call method on instance value:
    vector2d instance(1., 2.);
    (instance.*scale_ptr)(2.);

    /// Call method on instance via a pointer:
    vector2d* instance_ptr = new vector2d(3., 4.);
    (instance_ptr->*scale_ptr)(3.);
}
```

- С помощью оператора `::` доступа к пространству имён объекта можно также вытащить указатель на поле `&my_class::x`
- Лямбда выражения позволяют создавать функции без названия (но можно дать им название, сохранив в переменную типа `auto`)

Capture list

Return value type

This is a **closure lambda function** (expression)

function body

```
std::sort(
    vec.begin(),
    vec.end(),
    [] (const int first, const int second) -> bool {
        return first > second;
    });
```

- В листе захвата перед лямбдой можно давать лямбде доступ к переменным из внешнего пространства имён (и ссылкам на них) [&variable], также переписывая их [name_in_lambda = variable*2].
- При захвате по не по ссылке всё равно можно изменять переменный написав ключевое слово mutable

```
int num = 0;
std::function<void()> func = [num] () mutable -> void {
    /// Incrementing the local copy
    std::cout << ++ num << std::endl;
};
```

- Если нужно захватить константную ссылку, то нужно использовать [&name_in_lambda = std::as_const(variable)]
- Можно захватить всё пространство имён через [=] (как переменные) или [&] (как ссылки), но так лучше не делать, так как можно сломать много объектов в изначальном пространстве имён, передав лямбду в другое пространство (например функцию).
- В аргументах лямбды можно использовать auto. Это по сути делает её шаблоном, но неявным, поэтому использовать это нужно только когда вы точно не будете передавать неподходящие типы в неё.

НЕ ОК, подразумевалось использование template

```
auto multiply = [] (auto val) -> decltype(val) { return val * 2; };
std::cout << multiply(2) << " " << multiply(3.14) << std::endl;
```

- Или, начиная с 20ых плюсов, использовать явные шаблоны

```
auto forward_into_receiver =
    [&receiver = std::as_const(receiver)] <typename... args> (args&&... values) {
        receiver(std::forward<args>(values)...);
    };
```

- **std::function** создан для того, чтобы хранить и передавать лямбды или **любые** другие функциональные сущности в другие функции и методы и явно объявлять тип значений, которые они принимают и возвращают.

```
void receive_and_call (const std::function<void(int, float, double)>& f) {
    f(4, 2.31f, 3.14);
}
```

std::bind

std::mem_fn

std::reference_wrapper

std::invoke

Кратко описаны в презентации