5.19 Класс std::function, его основные методы и примеры использования. Опишите идею внутреннего устройства std::function: каким образом достигается возможность по ходу работы подменять хранящийся в ней функциональный объект? (Принимается любая работающая идея.)

std::function - это тип, который позволяет нам инициализировать себя любым объектом, который являтся callable.

Представьте, что у вас есть какой-то класс или какая-то функция, у которой одним из параметров должна быть другая функция. Например, мы пишем сортировку, и одним из ее параметров является компаратор. Но мы хотим, чтобы в качестве копаратора можно было передать не что угодно, а только вещи типа, которые вызываемы, то есь они callable, в терминах питона. Такие, что их можно вызыать от таких-то типов с такими-то параметрами

Ну, или например, мы пишем A*. И у нас там в качестве параметра используется какая-то функция (расстояния, эвристики). И эта функция должна быть полем класса. Какого типа должно быть это поле? Делать кучу шаблонных параметров на каждую функцию в классе очень странно, это некрасиво.

В c++11 появился такой тип, который называется std::function, который как раз позволяет нам хранить любой объект, имеющий оператор () именно от тех типов, которые мы ему сказали. Например, мы хотим, чтобы у нас была функция, принимающая два инта, возвращающая буль. Тогда пишем следующим образом:

```
std::function<bool(int, int)> f;
```

Этот объект можно проинициализировать любым объектом, имеющим оператор круглые скобки от двух int, возвращающий bool. В том числе и лямбда-функцией, в том числе указателем на сишную функцию

```
std::function<bool(int, int)> f;

f(1, 2);
```

Если мы пытаемся вызвать функцию, которая ничем не проницализировна, получаем исключение bad_function_call

```
f = [](int x, int y) {
    std::cout << "Hi!\n";
    return x < y;
}
```

Пример инициализации f

Здесь работает приведение типов. Если мы напишем так, то все сработает:

```
std::function<int(int, int)> f;

//f(1, 2);

f = [](int x, int y) {
    std::cout << "Hi!\n";
    return x < y;
};

f(1, 2);
```

```
std::function<int(bool, int)> f;

//f(1, 2);

f = [](int x, int y) {
    std::cout << "Hi!\n";
    return x < y;
};

f(1, 2);
```

```
struct S {

    bool operator()(int x, int y) const {
        std::cout << "Hello!\n";
        return x > y;
    }

};


int main() {

    std::function<bool(int, int)> f;

    //f(1, 2);

    f = [](int x, int y) {
        std::cout << "Hi!\n";
        return x < y;
    };

    f(1, 2);

    f = S();

}
```

Переприсваивание работает. Первоначальный объект уничтожится, а новый положится

```
bool g(int x, int y) {
    std::cout << "Blablabla!\n";
    return x == y;
}


int main() {

    std::function<bool(int, int)> f;

    //f(1, 2);

    f = [](int x, int y) {
        std::cout << "Hi!\n";
        return x < y;
    };

    f(1, 2);

    f = S();

    f(3, 4);

    f = &g;

```

Адрес сишной функции тоже скормится без проблем. Если убрать знак взятия адреса, то все тоже корректно будет работать

Внутренне устройство

Member functions

(constructor)	constructs a new std::function instance (public member function)
(destructor)	destroys a std::function instance (public member function)
operator=	assigns a new target (public member function)
swap	swaps the contents (public member function)
assign (removed in C++17)	assigns a new target (public member function)
operator bool	checks if a target is contained (public member function)
operator()	invokes the target (public member function)

Что внутри себя хранит function. Внутри нее есть некоторый класс Manager с шаблонным параметром F (тип функтора), в котором определены статические методы на каждое возможное действие с функтором (сделать копию, сделать инициализацию и т.д.).
Типы функторов:
• Указатель на C-style функцию;
• Обычный функциональный объект (т.е. объект, обладающий оператором () с соответствующими типами аргументов);
• Замыкание (то есть объект, созданный лямбда-выражением);

В Function хранятся указатели на эти методы (точнее, указатель на один метод с дополнительным параметром OperationType, с помощью которого можно выбирать нужный метод).

Когда мы инициализируем функцию чем-то новым, она вызывает у своего старого диспетчера (manager) метод "уничтожить", при этом, возможно, делается static_cast к нужному F, а потом создается указатель на нового диспетчера, и у него вызывается метод initialize (диспетчер запоминается)

Немного про идею реализации function

Класс Function должен обладать следующей функциональностью:
• Конструктор по умолчанию;
• Конструктор от Callable-объекта. Объект должно быть можно отдать в этот конструктор как в виде lvalue (и тогда Function должна скопировать его содержимое в себя), так и в виде rvalue (тогда Function должна мувнуть в себя его содержимое). Если принятый объект не является Callable с нужными аргументами, попытка создать Function от него должна приводить к CE.
• Конструктор копирования, конструктор перемещения, операторы присваивания (copy и move), деструктор.
• Оператор () с соответствующими аргументами, позволяющий вызвать хранимый в Function объект как функцию. Если там сейчас не хранится никакого объекта, нужно бросить исключение.

Кроме того:
• Function должна быть легковесным объектом. А именно, sizeof(Function) должен не превосходить 32 байт (ибо sizeof(std::function) в контесте именно такой).
• Предыдущий пункт означает, что если Callable-объект, который нужно сохранить в Function, достаточно большой, то под него надо выделять динамическую память. Это можно делать напрямую с помощью new/delete, аллокатором в этой задаче пользоваться необязательно.
• Обращения к new/delete надо по возможности экономить. Если новый Callable-объект можно положить на то же место, где лежал старый, то не надо делать перевыделение памяти.

• Для продвинутого потока: Если Callable-объект является обычным указателем на функцию, или указателем на метод, или чем-либо другим, по размеру не превосходящим max(void(*)(), void(C::*)()) (на практике это 16 байт), то динамическая память под него выделяться не должна! Такие объекты Function должна уметь хранить внутри своих полей, т.е. на стеке.
• Для основного потока: Если Callable-объект достаточно мал по размеру (не превосходит 16 байт), то динамическая память под него выделяться не должна! Такие объекты Function должна уметь хранить внутри своих полей, т.е. на стеке.