Лекция 6. Идиома RAII. Умные указатели.

Идиома RAII

Resource Acquisition Is Initialization (RAII) - идиома, которая подразумевает, что получение ресурса совмещено с инициализаций автоматической переменной на стеке. Таким образом происходит связывание времени жизни объекта с владением ресурсом.

Получение ресурса происходит в конструкторе объекта, при инициализации, в начале времени жизни объекта, а освобождение ресурса в деструкторе при уничтожении объекта. Такой подход предотвращает утечку ресурса, обеспечивая безопасность кода.

В качестве ресурса может выступать динамическая память, файловый дескриптор, мьютекс (что-то, что выдано третьей стороной во временное пользование, и что нужно вернуть по завершению использования)

Основная идея:

- Объект (автоматическая переменная на стеке) отвечает за жизненный цикл ресурса
- Ресурс захватывается в конструкторе объекта

- Ресурс освобождается в деструкторе объекта

Идиома RAII позволяет писать код без явных вызовов `delete` для выделенной памяти.

В стандартной библиотеке множество примеров реализации RAII:

- контейнеры, владеющие ресурсами (`std::vector`, `std::string`, `std::list`, ...)
- умные указатели `std::unique_ptr`, `std::shared_ptr`
- `std::lock_guard` для захвата мьютекса
- `std::fstream` для работы с файлами

В коде следует избегать явного управления ресурсами:

- вместо `new` - лучше использовать умные указатели
- вместо `new[]` - лучше использовать `std::vector` или другой подходящий контейнер

Умные указатели

Умный указатель - класс-обертка над сырым указателем, который управляет ресурсом (динамически выделенной памятью), и синтаксически может использоваться как обычный указатель, то есть имеет переопределение оператора разыменования `*` и оператора доступа по указателю `->`.

Умные указатели призваны решать проблемы при работе с динамической памятью:

- Утечки памяти (memory leak)
- Двойное удаление (double free)
- Использование неинициализированного указателя
- Висячий указатель (dangling pointer) и его использование
- Использование непарной версии оператора `delete`, `delete[]`

Простой **НЕ** рабочий вариант умного указателя:

```
using Resource = SomeClass;
class SmartPtr {
public:
    SmartPtr(Resource* raw_ptr) : raw_ptr_(raw_ptr) {};
    ~SmartPtr() { delete raw_ptr_; };
    Resource& operator*() const { return *raw_ptr_; };
    Resource* operator->() const { return raw_ptr_; };
private:
    Resource* raw_ptr_ = nullptr;
};
```

Использование умного указателя подразумевает что память выделяется непосредственно при конструировании объекта:

```
SmartPtr student_ptr(new Student("Student", 18));
```

- выделение памяти происходит непосредственно в конструкторе и объект `student_ptr` владеет ресурсом и сам следит за удалением объекта при выходе из области видимости

НЕ следует создавать сырой указатель, как-то с ним работать, передавать в другие функции, классы в сыром виде, а потом только конструировать умный указатель, так как это может привести к проблемам (например, висячему указателю).

```
Student* student_raw_ptr = new Student("Student", 18)
// do something with raw pointer
SmartPtr student_ptr(student_raw_ptr);
```

Проблема данной реализации в том, что при вызове конструктора копирования, который сгенерирует компилятор, произойдет копирование указателя и при выходе из области видимости будет ошибка double free из-за двойного удаления.

```
{
    SmartPtr student_ptr(new Student("Student", 18));
    SmartPtr bad_copy_student_ptr(student_ptr);
} // double free :(
```

Поэтому необходимо определиться с семантикой владения.

Можно выделить три семантики владения ресурсом:

- уникальное владение (один владелец, копирование невозможно, только перемещение)
- разделяемое владение (несколько владельцев, имеется счетчик ссылок на объект)
- не владеющее наблюдение (можно получить доступ к объекту, если он существует)

Умные указатели STL (C++11)

Умные указатели STL располагаются в заголовочном файле `<memory>`.

В стандартной библиотеки реализованы следующие умные указатели:

- `std::unique_ptr` - уникальное владение ресурсом
- `std::shared_ptr` - разделяемое владение ресурсом
- `std::weak_ptr` - слабое владение ресурсом (наблюдение)

Ранее существовал `auto_ptr`, но он *deprecated* начиная с C++11, и *removed* в C++17.
Им **НЕ** стоит пользоваться, так как в определенных ситуациях имеет проблемы с управлением ресурсом

Умный указатель `std::unique_ptr`

В стандартной библиотеке есть умный указатель `std::unique_ptr`, который эксклюзивно (единолично, уникально) владеет переданным ему ресурсом.

При создании умного указателя необходимо после имени класса указателя написать тип данных в угловых скобках `<>`, с которым будет работать указатель.
Если нужен указатель на тип `int`, то пишется `int`, писать `int*` **НЕ** нужно:

```
std::unique_ptr<int> ptr;
```

По умолчанию конструируется указатель с `nullptr`, но такой случай мало интересен:

```
std::unique_ptr<int> ptr;
std::unique_ptr<int> ptr{};
std::unique_ptr<int> ptr{nullptr};
std::unique_ptr<int> ptr = {};
std::unique_ptr<int> ptr = nullptr;
```

Принято создавать объект при получении ресурса (согласно **RAII**):

```
std::unique_ptr<int> item{new int{18}};
```

Поскольку владение уникальное, то класс указателя должно быть запрещено копировать, и, следовательно, у класса отсутствуют специальные методы для копирования.

```
SmartPtr(const SmartPtr&) = delete;
SmartPtr& operator=(const SmartPtr&) = delete;
```

Владение можно передать посредством перемещения:

```
std::unique_ptr<int> item{new int{18}};
std::unique_ptr<int> item_moved{std::move(item)}; // item = nullptr
```

Конструктор перемещения в таком случае может выглядеть примерно так:

```
SmartPtr(SmartPtr&& other) noexcept : raw_ptr_(other.raw_ptr_) {
    other.raw_ptr_ = nullptr; // other must be valid after move
}
```

Оператор присваивания перемещением корректно освобождает свой захваченный ресурс, перед перемещением (передачей владения) другого ресурса:

```
std::unique_ptr<int> item{new int{18}};
std::unique_ptr<int> item_moved{new int{19}};
item_moved = std::move(item); // item = nullptr
```

Оператор присваивания перемещением в таком случае может выглядеть примерно так:

```
SmartPtr& operator=(SmartPtr&& other) noexcept {
    if (raw_ptr_ != other.raw_ptr_) { // self assignment check
        delete raw_ptr_; // delete current resource
        raw_ptr_ = other.raw_ptr_; // copy resource pointer from other
        other.raw_ptr_ = nullptr; // other must be valid
    }
    return *this;
}
```

Умный указатель на массив

Умный указатель способен работать также с массивом объектов, при этом выделяемая память корректно освобождается с помощью парного оператора `delete[]`.

Для корректности освобождения памяти следует правильно указать тип `int[]`

```
std::unique_ptr<int[]> ptr{new int[10]{}};
```

НО считается более корректным подходом в таком случае использовать контейнер `std::vector<T>` вместо `std::unique_ptr<T[]>`

Методы и внешние функции

Помимо специальных методов, умный указатель имеет методы и внешние функции:

- `get()` - получить сырой указатель
- `release()` - освобождает владение (возвращает указатель и становится `nullptr`)
- `reset(T* ptr = nullptr)` - удаляет старый объект и принимает владение новым
- `swap(std::unique_ptr<T>&)` - обменивается владеющими указателями
- `get_deleter()` - возвращает удалитель
- Оператор `bool()` - приводит указатель к типу `bool`

Доступ к элементу в случае указателя на объект:

- Оператор `*` - разыменование указателя, возвращает ссылку
- Оператор `->` - доступ через указатель

Доступ к элементу в случае указателя на массив объектов:

- Оператор `[](size_t pos)` - доступ к элементу массива

Начиная с C++20 имеет оператор вывода в поток, который работает, как для сырого указателя, выводит адрес:

- Оператор `<<` - перегрузка для вывода в потом адреса сырого указателя

Внешние функции:

- Операторы сравнения - сравнивают сырые указатели (адреса)
- `std::make_unique` (C++14) - функция для создания умного указателя
- `std::make_unique_for_overwrite` (C++20) - функция для создания умного указателя без инициализации элемента (без вызова конструктора по умолчанию)
- `std::swap` - внешняя функция для работы в алгоритмах
- `std::hash` - хэш-функция для умного указателя

Передача в функцию

Обычно для передачи в функцию принято использовать `*` и передавать ссылку, или `get` и передавать сырой указатель, если известно, что функция ничего не делает с указателем.

- `func(SomeClass* ptr)` - не передает владение, но объекта может не существовать
- `func(const SomeClass& ptr)` - не передает владение, объект всегда существует

Функция может ожидать непосредственно `std::unique_ptr<SomeClass>`:

- `func(std::unique_ptr<SomeClass> ptr)` - передача владения
- `func(std::unique_ptr<SomeClass>& ptr)` - может изменить или освободить переданный умный указатель, например `std::swap` или метод `swap`
- `func(const std::unique_ptr<SomeClass>& ptr)` - несколько странно и вводит в заблуждение, `const` относится только к указателю, лучше передавать объект

Функция `std::make_unique` (C++14)

Создание объекта `std::unique_ptr` имеет ряд неудобств:

- Необходимо выделять память с помощью `new` явно
- Необходимо дважды писать тип: при выделении памяти и в качестве шаблонного параметра класса (а хочется использовать `auto`)
- Можно выделить память ранее, получить сырой указатель и использовать его даже после передачи в `std::unique_ptr` - Можно удалить память по сырому указателю, несмотря на передачу владения
- Нет гарантии безопасности относительно исключений при создании умного указателя непосредственно в аргументах функции

Последняя проблема самая важная. При создании умного указателя непосредственно в аргументах функции возможна утечка памяти, поскольку порядок конструирования аргументов **НЕ** определен и может возникнуть ситуация, когда память выделилась, но ещё не была помещена в умный указатель, и началось конструирование другого объекта при котором возникла исключительная ситуация (выброшено исключение). В таком случае вызов функции закончится с ошибкой и выделенная память **НЕ** будет освобождена.

```
void Func(std::unique_ptr<SomeClass> ptr, OtherClass obj);
OtherClass OtherThrowFunction();

Func(std::unique_ptr<SomeClass>(new SomeClass()), OtherThrowFunction());
```

- компилятор может сначала выделить память для `SomeClass`, а потом выполнить `OtherThrowFunction()` и только потом создать `std::unique_ptr<SomeClass>`

```
void Func(std::unique_ptr<SomeClass> ptr_one,
    std::unique_ptr<SomeClass> ptr_two);

Func(std::unique_ptr<SomeClass>(new SomeClass()),
    std::unique_ptr<SomeClass>(new SomeClass()));
```

- компилятор может сначала выделить память для каждого объекта `SomeClass`, и только потом создавать `std::unique_ptr<SomeClass>`, тогда в случае исключения в `SomeClass` один из объектов может утечь

В языке C++14 введена шаблонная функция `std::make_uniqe` (фабрика), которая лишена данных проблем. Функция `std::make_uniqe` принимает в качестве шаблонного параметра тип, а в качестве аргументов принимает параметры для конструкторов данного типа, и возвращает умный указатель. Это позволяет **НЕ** вызывать `new` явно, **НЕ** писать дважды тип, **НЕ** иметь в коде сырой указатель, гарантировать безопасность при исключениях

```
auto ptr = std::make_unique<SomeClass>(arg, other_arg);
```

Начиная с C++14 создавать умный указатель необходимо имеено таким образом

Также есть версия `std::make_unique_for_overwrite` (C++20), которая не вызывает конструктор по умолчанию, то есть не инициализирует память. Работает быстрее. Используется, когда после выделения объекта или массива объектов далее в коде производится заполнение.

Следует отметить, что `std::make_unique` не принимает нестандартный `Deleter`, в таких случаях необходимо писать свою шаблонную функцию.

Нестандартный *Deleter* `std::unique_ptr`

Помимо типа объекта `std::unique_ptr` имеет второй шаблонный параметр в `<>`. Вторым параметром в шаблон передается тип удалителя *Deleter*, а в аргументы непосредственно функция или функциональный объект.

По умолчанию указан удалитель, вызывающий `delete raw_ptr_` или `delete[] raw_ptr_` для массива.

Таким образом, для корректной работы класса `std::unique_ptr` с другими ресурсами, необходимо передать удалитель, который будет корректно освобождать ресурсы.

Например, при использовании функций языка C, когда необходимо вызывать `std::free`

```
std::unique_ptr<int, decltype(&std::free)> ptr(
    static_cast<int*>(std::malloc(sizeof(int))), std::free);
```

В целях логирования передать свой удалитель:

```
struct FreeDeleter {
    void operator()(void* ptr) const {
        std::cerr << "free() called\n";
        std::free(ptr);
    }
};

std::unique_ptr<int, FreeDeleter> ptr(
    static_cast<int*>std::malloc(sizeof(int))));

std::unique_ptr<int, FreeDeleter> ptr_explicit_deleter(
    static_cast<int*>(std::malloc(sizeof(int))), FreeDeleter{});
```

- если удалитель имеет конструктор по умолчанию, то компилятор способен сам его создать и нет необходимости передавать удалитель вторым аргументом

Если умный указатель используется не для выделения памяти, а для файлового дескриптора:

```
struct FileCloser {
    void operator()(FILE* file) const {
        if (file) {
            std::cerr << "Closing file\n";
            std::fclose(file);
        }
    }
};

std::unique_ptr<FILE, FileCloser> p2(std::fopen("test.txt", "r"));
```

- для этих нужд уже есть RAII обертка `std::fstream`

Также, если выделялась память с использованием пользовательского аллокатора, открывался сокет или другой ресурс.

Также удалитель может хранить состояние (поля).

Эффективность `std::unique_ptr`

Важно понимать, что `std::unique_ptr` очень легковесный класс-обертка над сырым указателем и его использование вместо сырого указателя практически **НЕ** замедляет выполнение кода и дает множество преимуществ.

Размер класса можно получить, используя оператор `sizeof`. В большинстве случаев использования он совпадает с размером указателя 8 байт на 64-разрядной платформы.

Поэтому, если есть возможность использовать `std::unique_ptr` среди прочих умных указателей, следует выбрать его, так как он практически **НЕ** имеет оверхеда (дополнительных накладных расходов) по производительности и по памяти.

Размер может увеличиваться, если принимается нестандартный *Deleter*, который может хранить состояние.

Умный указатель `std::shared_ptr`

В стандартной библиотеке есть умный указатель `std::shared_ptr`, который реализует разделяемое (совместное) владение динамически выделенным ресурсом.

Несколько экземпляров `shared_ptr` могут владеть одним и тем же объектом.

Данный указатель должен знать сколько имеется ссылок на конкретный объект, которым он владеет, чтобы при выходе из области видимости не удалять выделенную память (освобождать ресурс) преждевременно.

При выходе указателя из области видимости уменьшается счётчик ссылок. Освобождение ресурса происходит только тогда, когда из области видимости вышел последний указатель на объект (счетчик ссылок стал равным нулю).

Умный указатель `std::shared_ptr` устроен сложнее. Внутри хранится не только указатель на объект, но и указатель на блок управления (control block), который естественно также выделяется в динамической памяти. Блок управления хранит сразу два счетчика, счетчик сильных ссылок и счетчик слабых ссылок.

У данного указателя имеются специальные методы для копирования, при вызове которых происходит увеличение счетчика ссылок и простое копирование указателей.

Умный указатель на массив

До C++17 умный указатель не поддерживал работу с массивами по умолчанию.

Для корректности освобождения памяти массива необходимо передать пользовательский удалитель, работающий с массивом, вторым аргументом в конструктор. Удалитель у `shared_ptr` не является типом и храниться в блоке управления.

Начиная с C++17 появилась поддержка массивов и возможность обращения к элементу по индексу `[]`. Теперь по умолчанию вызывается корректный удалитель `delete raw_ptr_` или `delete[] raw_ptr_` для массивов

Методы и внешние функции

Помимо специальных методов, умный указатель имеет методы и внешние функции:

- `get()` - получить сырой указатель
- `reset(T* ptr = nullptr)` - удаляет старый объект и принимает владение новым, также имеет версии принимающие Deleter и Allocator
- `swap(std::shared_ptr<T>&)` - обменивается владеющими указателями
- `use_count()` - возвращает количество владельцев
- Оператор `bool()` - приводит указатель к типу `bool`

Доступ к элементу в случае указателя на объект:

- Оператор `*` - разыменование указателя, возвращает ссылку
- Оператор `->` - доступ через указатель

Доступ к элементу в случае указателя на массив объектов:

- Оператор `[](size_t pos)` (C++17) - доступ к элементу массива

Имеет оператор вывода в поток, который работает, как для сырого указателя, выводит адрес:

- Оператор `<<` - перегрузка для вывода в потом адреса сырого указателя

Внешние функции:

- Операторы сравнения - сравнивают сырые указатели (адреса)
- `std::make_shared` (C++11) - функция для создания умного указателя
- `std::make_shared_for_overwrite` (C++20) - функция для создания умного указателя без инициализации элемента (без вызова конструктора по умолчанию)
- `std::swap` - внешняя функция для работы в алгоритмах
- `std::hash` - хэш-функция для умного указателя
- `std::get_deleter` - возвращает указатель на удалитель
- `std::atomic` - поддержка атомарного доступа к умному указателю

Передача в функцию

Можно использовать `*` и передавать ссылку, или `get` и передавать сырой указатель, если известно, что функция **НЕ** сохраняет указатель или **НЕ** удаляет память или **НЕ** меняет указатель. Также важно гарантировать, что объект будет жив до окончания работы функции. Наиболее производительный вариант работы

- `func(SomeClass* ptr)` - не передает владение, но объекта может не существовать
- `func(const SomeClass& ptr)` - не передает владение, объект всегда существует

Функция может ожидать непосредственно `std::shared_ptr<SomeClass>`:

- `func(std::shared_ptr<SomeClass> ptr)` - разделение владения, увеличивает счетчик ссылок. Используется в асинхронных операциях, когда нужно гарантировать, что объект будет жив. Можно передать владение, используя `std::move(shared)` при передаче в функцию.
- `func(std::shared_ptr<SomeClass>& ptr)` - может изменить или освободить переданный умный указатель, например `std::swap` или метод `swap`, возврат через параметр
- `func(const std::shared_ptr<SomeClass>& ptr)` - наблюдение за объектом, счетчик ссылок не увеличивается, требует проверки на `nullptr` перед использованием. Может быть опасно и приводить к разыменованию `nullptr`
- `func(std::shared_ptr<SomeClass>&& ptr)` - явно указываем передачу владения

Функция `std::make_shared`

В отличие от `std::make_unique` (C++14), `std::make_shared` существует, начиная с C++11.

Функция `std::make_shared` принимает в качестве шаблонного параметра тип, а в качестве аргументов принимает параметры для конструкторов данного типа, и возвращает умный указатель.

Рекомендуется также создавать объект с помощью `std::make_shared`. Помимо решения ранее описанных проблем, функция `std::make_shared` выделяет память под объект и под блок управления единым фрагментом, в отличие от обычного создания умного указателя, при котором объект и блок управления могут находиться в памяти не рядом. Такой подход более оптимален с точки зрения производительности (адреса находятся близко) и меньше фрагментация памяти.

Таким образом `std::make_shared` имеет следующие преимущества:

- Одно выделение памяти вместо двух
- Лучшая локальность данных (объекта и блока управления)
- Гарантия безопасности при исключениях

НО в определенных ситуациях, когда объект занимает много памяти, а счетчик ссылок имеет слабую ссылку, невозможно освободить память под объект, пока **НЕ** обнулиться счетчик слабых ссылок.

Важным отличием `std::make_shared` от `std::make_unique` является то, что до C++20 данная функция **НЕ** могла работать с массивами (поддержка массивов в умный указатель `std::shared_ptr` была добавлена только в C++17)

Также есть версия `std::make_shared_for_overwrite` (C++20), которая не вызывает конструктор по умолчанию, то есть не инициализирует память. Работает

быстрее. Используется, когда после выделения объекта или массива объектов далее в коде производится заполнение.

Нестандартный *Deleter* `std::shared_ptr`

В отличие от `std::unique_ptr` умный указатель `std::shared_ptr` **НЕ** имеет второго шаблонного параметра для удалителя, то есть он **НЕ** является частью типа умного указателя. Удалитель можно передать аргументом при конструировании объекта.

По умолчанию при удалении вызывается `delete raw_ptr_` до C++17, начиная с C++17 также поддерживается версия для массивов, которая автоматически вызывает `delete[] raw_ptr_` для массивов.

Таким образом, для корректной работы класса `std::shared_ptr` с другими ресурсами, необходимо передать удалитель, который будет корректно освобождать ресурсы. Удалитель, как правило, храниться в блоке управления.

Пользовательский *Deleter* использует динамический полиморфизм, что означает обращение к *Deleter* через таблицу виртуальных функций

Создание `std::shared_ptr` из `std::unique_ptr`

Можно создавать `std::shared_ptr` из `std::unique_ptr` поскольку передается владение. При этом конструктор устроен более просто, создается только блок управления. Чтобы осуществить такую передачу, необходимо обернуть указатель `std::unique_ptr` в `std::move`

НЕЛЬЗЯ создавать `std::unique_ptr` из `std::shared_ptr`, поскольку есть блок управления, даже если владелец один, блок управления мог создаваться вместе с объектом в одном фрагменте памяти.

Эффективность `std::shared_ptr`

Важно понимать, что `std::shared_ptr` имеет значительно большие накладные расходы на использование, чем `std::unique_ptr`. Поэтому его использование должно быть только там, где действительно необходимо совместное владение.

Как для доступа к объекту, так и для доступа к блоку управления появляется второй уровень косвенности. К тому же сами счетчики являются атомарными `std::atomic<size_t>`, и их изменение производится несколько медленнее, чем обычных переменных. Но это необходимо для безопасного изменения с разделяемым владением, при возможности изменений из разных потоков.

Размер класса можно получить, используя оператор `sizeof`. Класс имеет размер 16 байт, то есть совпадает с размером двух указателей 8 байт для 64-разрядной

платформы.

Размер контрольного блока может увеличиваться, если принимается нестандартный *Deleter*, который может хранить состояние.

В случае нестандартного удалителя также снижается эффективность, так как вызов идет через таблицу виртуальных функций.

Проблема циклических ссылок `std::shared_ptr`

Иногда использование умного указателя `std::shared_ptr` может приводить к утечкам памяти. Эта проблема называется проблемой циклических (перекрестных) ссылок. Проблема возникает, когда два объекта какого-то класса, хранят умные указатели `std::shared_ptr` друг на друга.

В такой ситуации при выходе из области видимости каждого из объектов не произойдет обнуления счетчика ссылок, поскольку остается объект, который имеет ссылку. Следовательно, выделенная память не освободится, произойдет утечка.

```
struct Priest {
    std::shared_ptr<Dog> dog;
};

struct Dog {
    std::shared_ptr<Priest> owner;
};

void problem_cyclic_ref() {
    auto priest_ptr = std::make_shared<Priest>(); // priest_shared_count = 1
    auto dog_ptr = std::make_shared<Dog>(); // dog_shared_count = 1

    priest_ptr->dog = dog_ptr; // dog_shared_count = 2
    dog_ptr->owner = priest_ptr; // priest_shared_count = 2
} // dtors for Dog and Priest doesn't call because counters = 1 (memory leak)
```

Важно понимать, что это **НЕ** обязательно прямая ссылка владения, куда чаще это может происходить в списке или в графе с циклами через несколько элементов

Для решения этой проблемы как раз существуют слабый указатель `std::weak_ptr`

Умный указатель `std::weak_ptr`

Для решения проблемы циклических ссылок в стандартной библиотеки имеется слабый (не владеющий) указатель `std::weak_ptr`. Умный указатель `std::weak_ptr` является наблюдателем за владеющим указателем `std::shared_ptr`.

Указатель `std::weak_ptr` позволяет безопасно узнать, существует ли объект, на который он ссылается, и получить к нему временный доступ, если объект всё ещё жив. Наличие слабой ссылки не продлевает объекту жизнь. Он будет удалён, как только на него перестанут ссылаться `std::shared_ptr`.

Блок управления, создаваемый `std::shared_ptr` имеет не только счетчик владеющих ссылок, но также счетчик слабых ссылок, что позволяет знать, когда можно освобождать память выделенную непосредственно под блок управления.

Указатель `std::weak_ptr` создается из указателя `std::shared_ptr` или из другого `std::weak_ptr`. При этом увеличивается счетчик слабых ссылок в блоке управления.

Указатель `std::weak_ptr` хранит аналогично `std::shared_ptr` два указателя, на данные и на блок управления

Методы и внешние функции

Помимо специальных методов, умный указатель имеет методы и внешние функции:

- `reset()` - убирает ссылку на владеющий объект
- `swap(std::weak_ptr<T>&)` - обменивается указателями на наблюдаемый объект
- `use_count()` - возвращает количество владельцев исходного объекта
- `expired()` - проверяет, истекло ли время жизни исходного объекта `use_count() == 0`
- `lock()` - создает владеющий указатель `std::shared_ptr` на объект, если он ещё жив

Внешние функции:

- `std::swap` - внешняя функция для работы в алгоритмах
- `std::atomic` - поддержка атомарного доступа к умному указателю

Передача в функцию

Функция принимает `std::weak_ptr<SomeClass>`:

- `func(std::weak_ptr<SomeClass> ptr)` - создание копии, увеличивает счетчик ссылок.
- `func(std::weak_ptr<SomeClass>& ptr)` - может изменить или освободить переданный умный указатель, например `std::swap` или метод `swap`
- `func(const std::weak_ptr<SomeClass>& ptr)` - счетчик ссылок не увеличивается
- `func(std::weak_ptr<SomeClass>&& ptr)` - явно забираем `std::weak_ptr`, исходный `std::weak_ptr` будет `expired`