

# Умные указатели

Владислав Хорев  
Ведущий программист в компании Andersen



# Проверка связи





## Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти



## Поставьте в чат:

-  если меня видно и слышно
-  если нет

# Владислав Хорев

О спикере:

- Ведущий программист в компании Andersen
- Работает в IT с 2011 года
- Опыт разработки на C++ более 11 лет



# Вспоминаем прошрое занятие

**Вопрос:** чем bidirectional итераторы  
отличаются от forward?



# Вспоминаем прошрое занятие

**Вопрос:** чем bidirectional итераторы отличаются от forward?

**Ответ:** тем, что можно итерироваться в 2 стороны



# Вспоминаем прошное занятие

**Вопрос:** какие 2 типа итераторов  
гарантированно предоставляет каждый  
контейнер из stl?



# Вспоминаем прошрое занятие

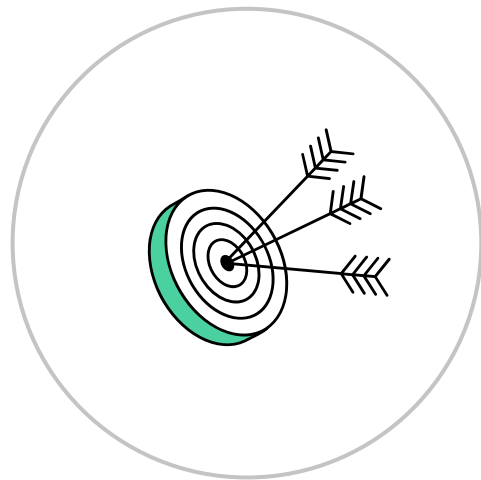
**Вопрос:** какие 2 типа итераторов  
гарантированно предоставляет каждый  
контейнер из stl?

**Ответ:** `container::iterator`,  
`container::const_iterator`



# Цели занятия

- Узнаем, зачем нужны умные указатели
- Узнаем, какие бывают указатели и разберем основные сценарии использования

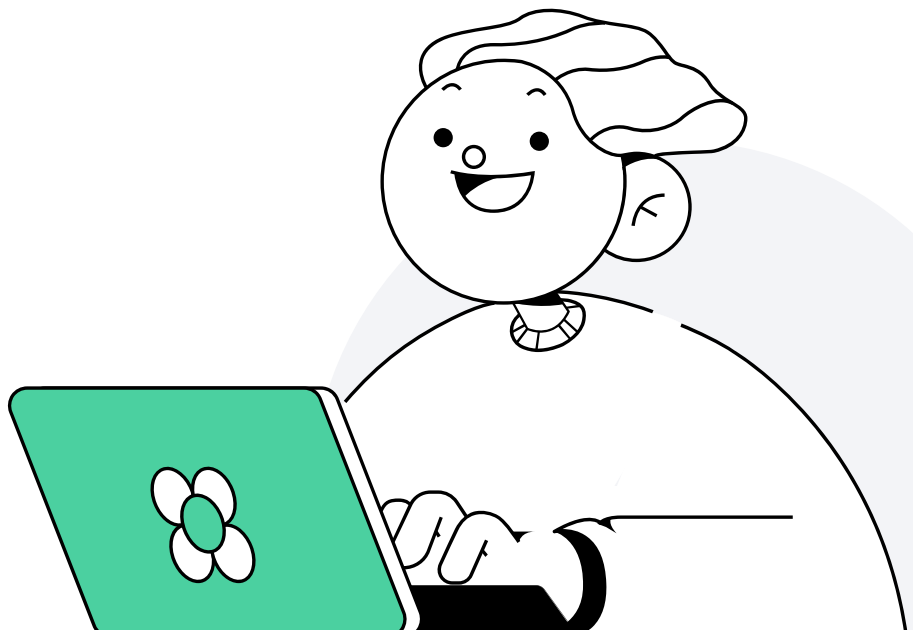




# План занятия

- 1 Виды умных указателей
- 2 `std::unique_ptr`
- 3 `std::shared_ptr`
- 4 `std::weak_ptr`
- 5 Домашнее задание

\*Нажми на нужный раздел для перехода



# Зачем это нужно?

Посмотрите на следующий код. Какие недостатки вы видите?

```
int some_func() {  
    int* my_data = new int[600];  
    // Проверка какого-либо условия  
    if (some_condition) {  
        // что-то делаем  
        return 1;  
    }  
    // Работаем дальше  
    if (another_condition) {  
        test_func();  
        return 2;  
    }  
    // Удаляем память  
    delete[] my_data;  
}
```

# Зачем это нужно?

- Если выполнится какое-либо из условий *some\_condition, another condition*, то выделенная память не удалится. Нужно в каждом условии писать удаление — это дублирование кода
- Если вдруг в функции *test\_func* возникнет исключение, то опять память не удалится, и возникнут утечки памяти

## Что требуется?

Иметь объект, который самостоятельно заботится о высвобождении своих ресурсов

# Виды умных указателей



1

# Умные указатели

Для этого в std введены следующие классы:

## `auto_ptr`

Старый умный указатель — **не использовать!**  
(Удалён в C++17)

## `unique_ptr`

Предоставляет  
уникальное  
владение объектом

## `shared_ptr`

Предоставляет  
совместное  
владение объектом

## `weak_ptr`

Решает некоторые  
проблемы, которые  
возникают с  
`shared_ptr`

# std::auto\_ptr

auto\_ptr был удалён в 17 стандарте и вот почему:

Синтаксис создания: **std::auto\_ptr<T> ptr(new T(...))**

```
std::auto_ptr<int> object1(new int(10)); // 1 объект  
std::auto_ptr<int> object2(new int(5)); // 2 объект  
object1 = object2; // разрушающее копирование
```

**Вопрос:** Что происходит в последней строчке?

# std::auto\_ptr

auto\_ptr был удалён в 17 стандарте и вот почему:

Синтаксис создания: **std::auto\_ptr<T> ptr(new T(...))**

```
std::auto_ptr<int> object1(new int(10)); // 1 объект
std::auto_ptr<int> object2(new int(5)); // 2 объект
object1 = object2; // разрушающее копирование
```

**Вопрос:** Что происходит в последней строчке?

**Ответ:** Происходит передача владения object2 в object1. При этом object2 становится невалидным — пустым. Такого поведения хочется избежать.

На замену ему ввели указатель unique\_ptr.

# std::unique\_ptr



2





**std::unique\_ptr воплощает в себе семантику уникального владения:**

- **у базового указателя только один владелец,**
- **копирование std::unique\_ptr не разрешается, доступна только передача владения**

# std::unique\_ptr

```
void use_raw_pointer()
{
    my_class* ptr = new my_class();

    // использование сырого указателя ptr – потенциальные утечки памяти

    // не забываем удалить
    delete ptr;
}

void use_unique_pointer()
{
    // умный указатель
    std::unique_ptr<my_class> ptr(new my_class());

    // использование полей
    auto field_value = ptr->field1;

} // ptr автоматически удалится здесь
```

Умные указатели  
соответствуют идиоме RAII!

# std::unique\_ptr

Создание объекта можно выполнять 2 способами:

1. через конструктор (**std::unique\_ptr<T> ptr(new T(...))**)

```
std::unique_ptr<my_class> ptr(new my_class("string data", 1));
```

# std::unique\_ptr

Создание объекта можно выполнять 2 способами:

## 2. через функцию **std::make\_unique<T>(...)**

Этот вариант считается предпочтительнее, так он:

- красивее с точки зрения современного C++ (нет оператора new)
- безопаснее с точки зрения гарантии исключений

Однако, он не всегда подойдет, например, если нужно использовать кастомные делитеры:

```
auto ptr = std::make_unique<my_class>("string data", 1);
```

**std::unique\_ptr поддерживает работу с массивами!**

# std::unique\_ptr

unique\_ptr нельзя копировать, можно только передавать владение с помощью функции std::move

```
auto ptr1 = std::make_unique<some_class>(10, 5.);  
auto ptr2 = std::make_unique<some_class>(10, 1.);  
  
//ptr1 = ptr2; // не скомпилируется  
ptr1 = std::move(ptr2);  
// ptr1 указывает на вектор из десяти 1, а в ptr2 ресурсы освобождены  
  
take_ownership(std::move(ptr1)) // передача владения в функцию
```

А в остальном, с ними можно оперировать как с обычными указателями:

- доступна операция разыменования \*
- доступ к полям через оператор ->

# std::unique\_ptr

**Вопрос:** Что будет при запуске этого фрагмента кода?

```
some_class *obj = new some_class();  
std::unique_ptr<some_class> ptr1(obj);  
std::unique_ptr<some_class> ptr1(obj);
```



# std::unique\_ptr

**Вопрос:** Что будет при запуске этого фрагмента кода?

```
some_class *obj = new some_class();  
std::unique_ptr<some_class> ptr1(obj);  
std::unique_ptr<some_class> ptr1(obj);
```

**Ответ:** ptr1 и ptr2 попытаются удалить obj, что приведет к неопределенному поведению.

Аналогично не стоит использовать умные указатели с оператором delete!



# std::unique\_ptr

В следующем примере показан пример полиморфизма с помощью умных указателей:

```
class base
{
public:
    base() { std::cout << "base constructor\n"; }
    virtual ~base() = default;
    virtual void func() { std::cout << "func in base class\n"; }
};

struct derived : public base
{
    derived() { std::cout << "derived constructor\n"; }

    void func() override { std::cout << "func in derived class\n"; }
};

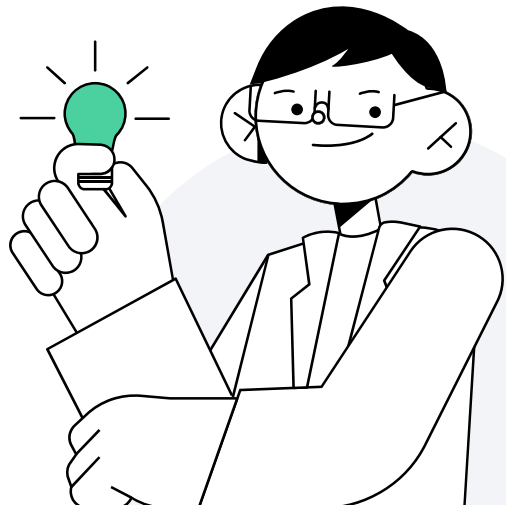
int main()
{
    std::unique_ptr<base> ptr = std::make_unique<derived>(); //храним указатель на класс потомок в базовом
    ptr->func(); // выведет func in derived class
    return 0;
}
```



# std::unique\_ptr

Функции, которые пригодятся в работе:

Метод	Описание
release	Возвращает указатель на управляемый объект и освобождает владение
reset	Заменяет управляемый объект
swap	Обменивает управляемые объекты



# std::shared\_ptr



3

The diagram illustrates the internal structure of a `std::shared_ptr`. It consists of two overlapping circles. The left circle is white with a black border and contains the number '3' in black, representing the reference count. The right circle is light blue with a black border and is partially cut off by the edge of the frame.



**std::shared\_ptr воплощает в себе семантику совместного владения:**

- **все указатели, указывающие на объект, сотрудничают для обеспечения гарантии, что его уничтожение произойдет в точке, где он станет более не нужным**

# Как это работает?

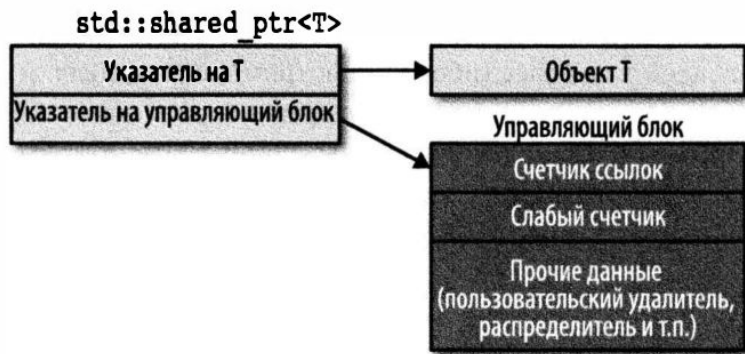
Каким образом реализовать такое поведение?

Очевидно, что нужно где-то хранить какой-то счетчик, который увеличивается, когда мы создаем ссылку на наш объект (разделяем владение нашим ресурсом) и уменьшается, когда ссылки больше не используются.

# Как это работает?

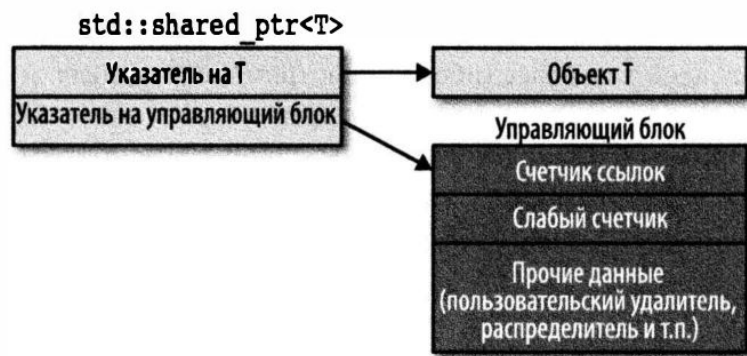
Счетчик ссылок и другая дополнительная информация хранятся в контрольном блоке.

В самом `shared_ptr` хранится только указатель на объект, ресурсами которого мы управляем, и указатель на контрольный блок.



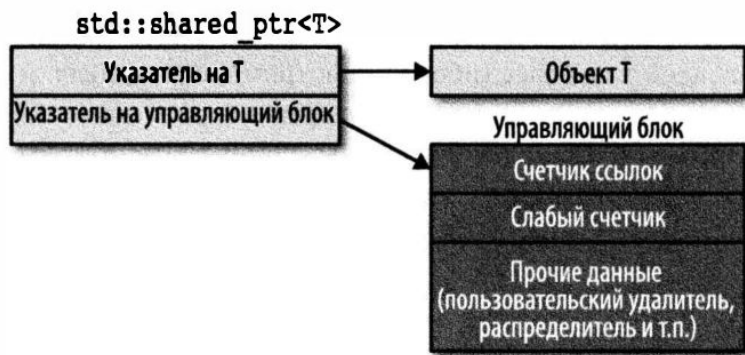
# Как это работает?

**Вопрос:** Почему нельзя хранить счетчик ссылок в самом объекте `std::shared_ptr`?



# Как это работает?

**Вопрос:** Почему нельзя хранить счетчик ссылок в самом объекте `std::shared_ptr`?



**Ответ:** Потому, что `shared_ptr` можно скопировать, а исходный `shared_ptr` уже может “умереть” и информация из контрольного блока станет недоступна.

# std::shared\_ptr

Создание объекта можно выполнять несколькими способами:

1. через конструктор (**std::shared\_ptr<T> ptr(new T(...))**)

```
std::shared_ptr<my_class> ptr(new my_class("string data", 1));
```



# std::shared\_ptr

Создание объекта можно выполнять несколькими способами:

## 2. через функцию **std::make\_shared<T>(...)**.

Этот вариант считается предпочтительнее, так он:

- красивее с точки зрения современного C++ (нет оператора new)
- безопаснее с точки зрения гарантии исключений

```
auto ptr = std::make_shared<my_class>("string data", 1);
```

# std::shared\_ptr

Пример принципа работы shared\_ptr:

```
int main()
{
    auto ptr = std::make_shared<base>(); // счетчик ссылок = 1
    {
        auto ptr1 = ptr; // счетчик ссылок = 2
        // do-something
    }
    // счетчик ссылок = 1

    return 0;
} // счетчик ссылок = 0, объект уничтожается
```

# std::shared\_ptr

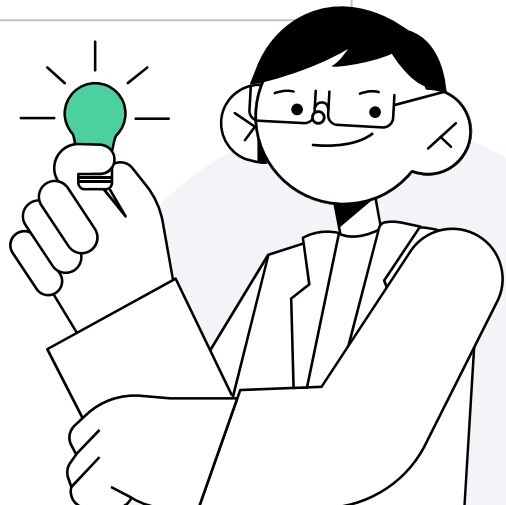
Неправильное использование std::shared\_ptr может привести к двойному высвобождению ресурсов:

```
int main()
{
    some_class *obj = new some_class();
    auto ptr = std::shared_ptr<some_class>(obj);
    {
        auto ptr1 = std::shared_ptr<some_class>(obj);
        // do-something
    }
    return 0;
}
```

# std::shared\_ptr

Функции, которые пригодятся в работе:

Метод	Описание
use_count	Возвращает количество объектов shared_ptr, ссылающихся на один и тот же управляемый объект
reset	Заменяет управляемый объект
swap	Обменивает управляемые объекты



# std::weak\_ptr



4

# std::shared\_ptr: простая циклическая зависимость

Однако, бывают ситуации, когда shared\_ptr использовать проблемно.

Рассмотрим упрощенную циклическую зависимость:

```
class test_class
{
public:
    std::shared_ptr<test_class> m_ptr;
    ~test_class() { std::cout << "destructor called\n"; }
};

int main()
{
    {
        auto ptr = std::make_shared<test_class>();
        ptr->m_ptr = ptr;
        std::cout << ptr.use_count(); // количество ссылок = 2: ptr ссылается сам на себя же
    } // количество ссылок стало 1, ресурс не освободится
    return 0;
}
```

# std::shared\_ptr: замена на weak\_ptr

Решение: замена члена класса на weak\_ptr.

Это поможет, так как weak\_ptr не считается владеющим указателем!

```
class test_class
{
public:
    std::weak_ptr<test_class> m_ptr;
    ~test_class() { std::cout << "destructor called\n"; }
};

int main()
{
    {
        auto ptr = std::make_shared<test_class>();
        ptr->m_ptr = ptr;
        std::cout << ptr.use_count(); // количество ссылок = 1
    } // ресурсы освободятся
    return 0;
}
```

# std::shared\_ptr: пересечение указателей

```
class node
{
public:
    int m_value;
    std::shared_ptr<node> parent;
    node(int value) : m_value{ value } {};
    ~node() { std::cout << "destructor called\n"; }
};

int main()
{
    {
        auto node1 = std::make_shared<node>(1); // 1ая ссылка на node1
        auto node2 = std::make_shared<node>(2); // 1ая ссылка на node2
        node1->parent = node2; // 2ая ссылка на node2
        node2->parent = node1; // 2ая ссылка на node1
    } // ни один из выделенных ресурсов не освободится

    return 0;
}
```





**std::weak\_ptr разработан для решения проблемы циклической зависимости.**

**Он является наблюдателем: получает доступ к тому же объекту, на который указывает std::shared\_ptr, но не является владельцем**

# std::weak\_ptr

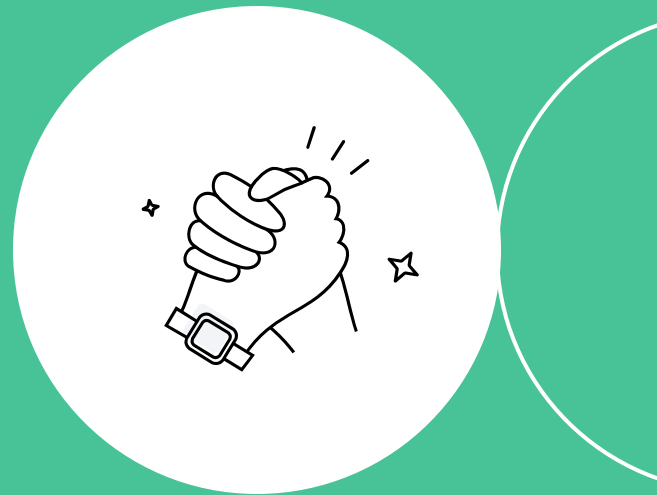
weak\_ptr нельзя использовать напрямую, его необходимо конвертировать в std::shared\_ptr с помощью метода lock():

```
class node
{
public:
    int m_value;
    std::weak_ptr<node> parent;
    node(int value) : m_value{ value } {};
    ~node() { std::cout << "destructor called\n"; }
};

int main()
{
    {
        auto node1 = std::make_shared<node>(1);
        auto node2 = std::make_shared<node>(2);
        node1->parent = node2;
        node2->parent = node1;

        //node1->parent->m_value = 4; // не скомпилируется
        node1->parent.lock()->m_value = 4;
    }
    return 0;
}
```

# Итоги



# Итоги занятия

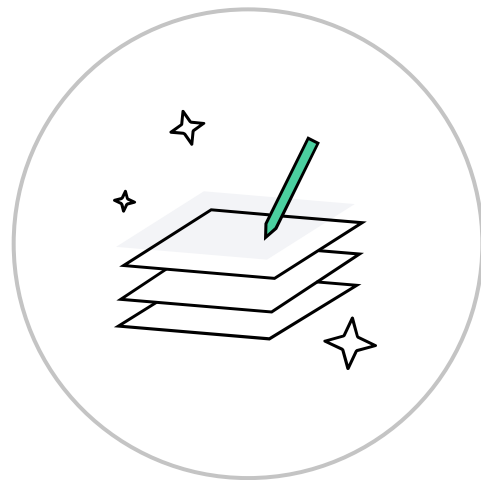
- 1 Узнали, зачем нужны умные указатели
- 2 Изучили умные указатели из STL: `unique_ptr`, `shared_ptr`, `weak_ptr` и разобрали основные сценарии использования



# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- 1 Вопросы по домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



# Дополнительные материалы

- [Документация по умным указателям](#)
- Мейерс Скотт. Эффективный и современный C++:  
42 рекомендации по использованию C++11 и C++14



**Задавайте вопросы  
и пишите отзыв о лекции**

