HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 03 – Продолжаем про move семантику

Глава 1

Механика перемещения и специальные функции класса

Современный C++ (начиная с C++11) ввел концепцию Move Semantics (семантика перемещения), которая фундаментально изменила подход к управлению ресурсами. Если раньше передача тяжелого объекта (например, `std::vector` на гигабайт данных) означала глубокое копирование, то теперь мы можем передать владение ресурсом за константное время. В этой главе мы детально разберем, как реализовать перемещающие операции, как избежать распространенных ошибок и как компилятор генерирует (или удаляет) специальные функции класса.

Анатомия Move Assignment Operator

Оператор перемещающего присваивания (`move assignment operator`) — это специальная функция-член класса, которая вызывается, когда объекту присваивается *rvalue* того же типа. В отличие от копирующего присваивания, цель этой операции — не дублировать ресурс, а *украсть* его у временного объекта, оставив последний в состоянии, пригодном для безопасного уничтожения.

Паттерн реализации идиомы copy-and-swap

Один из классических способов реализации операторов присваивания — идиома **copy-and-swap** (в контексте move — скорее *move-and-swap*). Она выглядит следующим образом:

```
class Holder {
public:
    // ... конструкторы ...

    // Move assignment
    Holder& operator=(Holder&& other) noexcept {
        // Меняем местами текущее состояние с состоянием other
        swap(*this, other);
        return *this;
    }

private:
    void swap(Holder& a, Holder& b) {
```

```
        // ... реализация обмена полей ...
    }
};
```

На заметку

Механика: При вызове `swap`, текущий ресурс объекта (который нужно уничтожить) перемещается в `other`, а ресурс из `other` перемещается в `*this`. Поскольку `other` — это *rvalue*, который скоро будет уничтожен, он заберет старый ресурс нашего объекта с собой в могилу (вызовет деструктор).

Однако, у этого подхода есть недостатки:

1. **Лишняя работа:** `swap` выполняет три перемещения (A -> tmp, B -> A, tmp -> B). Для простого присваивания это может быть избыточно.
2. **Отложенное уничтожение:** Старый ресурс уничтожается не сразу, а только когда будет вызван деструктор `other`. Это может быть критично, если ресурс удерживает, например, файловый дескриптор или мьютекс.

Оптимизированная реализация через std::exchange

Более современный и эффективный подход — использование `std::exchange` (C++14). Эта функция заменяет значение объекта новым и возвращает старое. Это позволяет реализовать перемещение "в одну строчку" с четкой семантикой передачи владения.

```
#include <utility> // для std::exchange

class Buffer {
    char* data_ = nullptr;
    size_t size_ = 0;

public:
    // ...

    Buffer& operator=(Buffer&& other) noexcept {
        if (this == &other) {
            return *this; // Защита от self-assignment
        }

        // 1. Освобождаем текущий ресурс
        delete[] data_;

        // 2. Забираем ресурс у other и "зануляем" его
        data_ = std::exchange(other.data_, nullptr);
        size_ = std::exchange(other.size_, 0);

        return *this;
    }
};
```

В этом коде `std::exchange(other.data_, nullptr)` делает следующее:

1. Читает текущее значение `other.data_`.
2. Присваивает `other.data_` значение `nullptr`.
3. Возвращает прочитанное (старое) значение, которое мы записываем в `this→data_`.

Это устраняет промежуточные свопы и гарантирует, что объект-источник (`other`) останется в пустом состоянии.

Проблема Self-Assignment (Самоприсваивание)

В копирующем присваивании проверка `if (this == &other)` обязательна, чтобы не удалить свой собственный ресурс перед копированием. В перемещающем присваивании ситуация тоньше.

> **Важно!**
>
> Если вы используете реализацию через `swap`, самоприсваивание безопасно (своп объекта с самим собой ничего не ломает), но бесполезно тратит такты процессора. Если вы используете прямую реализацию с `delete[] data_`, то без проверки `if (this == &other)` вы удалите ресурс, который собираетесь переместить в себя же -> **Undefined Behavior** (use-after-free).

Хотя самоприсваивание rvalue (написание `x = std::move(x)`) — это редкая и странная операция, корректная реализация должна её обрабатывать. Стандартная библиотека требует, чтобы объекты были устойчивы к самоприсваиванию.

Концепция "Moved-from state"

Когда мы перемещаем данные из объекта (источника), он остается в состоянии, которое стандарт называет "**valid but unspecified state**" (валидное, но неопределенное).

> **Moved-from state**
>
> Объект, из которого переместили данные, должен находиться в таком состоянии, что:
>
> - Его деструктор отработает корректно (не упадет, не сделает double free).
> - Ему можно присвоить новое значение (оператор присваивания сработает корректно).
> - Значение его полей не гарантировано (если явно не оговорено классом).

Рассмотрим пример "плохого" перемещения:

```
class BadString {
    char* str;
public:
    BadString(BadString&& other) {
        // Мы скопировали указатель...
        this->str = other.str;
        // ...НО ЗАБЫЛИ занулить other.str!
    }

    ~BadString() {
```

```
        delete[] str;
    }
};
```

В этом случае, когда `other` выйдет из области видимости, его деструктор удалит память по адресу `str`. Позже деструктор нашего нового объекта снова попытается удалить `str`. Это классический **double free**.

Правило: **Всегда оставляйте источник в состоянии, безопасном для разрушения (обычно это null pointers, zero size).**

Rule of 5 vs Rule of 0

C++11 расширил "Правило трех" (Rule of 3) до "Правила пяти" (Rule of 5) из-за добавления семантики перемещения.

1. **Деструктор**
2. **Копирующий конструктор**
3. **Копирующий оператор присваивания**
4. **Перемещающий конструктор**
5. **Перемещающий оператор присваивания**

Rule of 5

Если классу требуется ручное управление ресурсом (например, сырой указатель, дескриптор файла), вы, скорее всего, должны реализовать (или явно объявить) все 5 специальных функций.

```
class ResourceManager {
    Resource* res;
public:
    ~ResourceManager() { delete res; } // 1

    ResourceManager(const ResourceManager& other) { ... } // 2
    ResourceManager& operator=(const ResourceManager& other) { ... } // 3

    ResourceManager(ResourceManager&& other) noexcept { ... } // 4
    ResourceManager& operator=(ResourceManager&& other) noexcept { ... } // 5
};
```

Rule of 0

Если ваш класс не управляет ресурсами напрямую, а использует RAII-обертки (`std::string`, `std::vector`, `std::unique_ptr`), вам **не нужно** писать ни одну из специальных функций. Компилятор сгенерирует их автоматически и корректно.

> **Важно!**
>
> **Rule of 0** — предпочтительный подход. Это принцип "Пиши меньше кода". Если вы можете выразить семантику класса через комбинацию стандартных типов, делайте это.

Implicit Deletion (Неявное удаление)

Правила генерации специальных функций в C++ довольно запутаны. Одно из важнейших правил:

> **На заметку**
>
> Если вы объявляете (или определяете) любую перемещающую операцию (конструктор или оператор присваивания), то **копирующие операции неявно удаляются** (становятся `delete`).

Это сделано для безопасности: если класс поддерживает перемещение, скорее всего, он владеет уникальным ресурсом (как `unique_ptr`), который нельзя просто скопировать.

Пример ловушки:

```
class Widget {
public:
    Widget(Widget&&) noexcept; // Пользовательский move ctor
    // Copy ctor неявно удален!
    // Copy assignment неявно удален!
    // Move assignment не объявлен (но и не удален, просто не сгенерирован)
};

void func() {
    Widget w1;
    Widget w2 = w1; // ОШИБКА КОМПИЛЯЦИИ: call to deleted constructor
}
```

Если вы хотите вернуть копирование, вы должны объявить его явно: `Widget(const Widget&) = default;`

noexcept в перемещающих операциях

Ключевое слово `noexcept` критически важно для производительности перемещения, особенно при работе со стандартными контейнерами, такими как `std::vector`.

Проблема транзакционности вектора

Рассмотрим, что происходит, когда `std::vector` исчерпывает свою емкость (`capacity`) и должен расшириться: 1. Выделяется новый, больший буфер памяти. 2. Элементы переносятся из старого буфера в новый. 3. Старый буфер удаляется.

В C++98 элементы всегда копировались. Если при копировании N-го элемента возникало исключение, вектор просто уничтожал уже созданные копии в новом буфере и освобождал

его. Старый буфер оставался нетронутым. Это обеспечивало **Strong Exception Guarantee** (строгую гарантию исключений): операция либо выполняется полностью, либо не меняет состояние программы.

В C++11 мы хотим перемещать элементы (это дешевле). Но если перемещающий конструктор кинет исключение на N-м элементе, мы оказываемся в беде:

- Часть элементов уже перемещена в новый буфер.
- Их "оригиналы" в старом буфере теперь находятся в *moved-from state* (по сути, разрушены или пусты).
- Мы не можем "откатить" операцию, потому что обратное перемещение тоже может кинуть исключение!
- Данные безвозвратно потеряны.

std::move_if_noexcept

Чтобы сохранить Строгую гарантию исключений, `std::vector` использует утилиту `std::move_if_n`
Логика при реаллокации следующая:

- Если перемещающий конструктор типа элемента помечен как `noexcept`, вектор использует перемещение.
- Если `noexcept` нет, вектор **откатывается к копированию** (использует copy constructor), даже если доступен move constructor.
- Если тип *move-only* (как `unique_ptr`) и не `noexcept`, вектор не дает гарантий безопасности исключений (или не компилируется, в зависимости от реализации).

> **Важно!**
>
> Если вы забыли написать `noexcept` у перемещающего конструктора, ваш код будет компилироваться и работать, но `std::vector<MyType>` будет работать **медленно**, выполняя глубокое копирование при каждом расширении. Это "тихий убийца производительности".

```
class FastWidget {
public:
    // Обязательно noexcept!
    FastWidget(FastWidget&&) noexcept { /* ... */ }

    // Аналогично для присваивания
    FastWidget& operator=(FastWidget&&) noexcept { /* ... */ }
};
```

Техническое отступление: noexcept оператор

В C++ `noexcept` — это не только спецификатор, но и оператор, который возвращает `bool` на этапе компиляции. Он проверяет, может ли выражение теоретически кинуть исключение.

```
template <typename T>
class Wrapper {
```

```
    T value;
public:
    // Move ctor является noexcept ТОЛЬКО если T имеет noexcept move ctor
    Wrapper(Wrapper&& other) noexcept(std::is_nothrow_move_constructible_v<T>)
        : value(std::move(other.value)) {}
};
```

Это позволяет создавать гибкие шаблоны, которые транслируют гарантии исключений вложенных типов.

Глава 2

Универсальные ссылки и идеальная передача (Perfect Forwarding)

В предыдущей главе мы обсуждали rvalue-ссылки (`T&&`) как инструмент для реализации семантики перемещения. Однако в шаблонах C++ синтаксис `T&&` приобретает совершенно иное значение. Это одна из самых запутанных тем для новичков: одни и те же символы могут означать "rvalue-ссылка" или "универсальная ссылка" (Universal Reference), в зависимости от контекста.

В этой главе мы разберем механику вывода типов, правила схлопывания ссылок (Reference Collapsing) и современные способы работы с квалификаторами значений, включая нововведения C++23.

Универсальные ссылки (Forwarding References)

Термин "Universal Reference" был введен Скоттом Мейерсом, но в стандарте C++ он называется **Forwarding Reference**. Это ссылка, которая может вести себя и как lvalue-ссылка, и как rvalue-ссылка, в зависимости от того, чем она инициализирована.

> **Универсальная ссылка**
>
> Ссылка является универсальной только при выполнении двух условий одновременно:
>
> 1. Вывод типа (*type deduction*) происходит именно для этой переменной.
> 2. Переменная объявлена строго как `T&&` (где `T` — имя шаблонного параметра).

Рассмотрим разницу на примерах:

```
// 1. Обычная Rvalue-ссылка
void func(Widget&& w); // Тип Widget конкретен, вывода типов нет.
                       // Принимает ТОЛЬКО rvalue.

// 2. Универсальная ссылка
template <typename T>
void wrapper(T&& arg); // T выводится компилятором.
                       // Принимает И lvalue, И rvalue.

```

```cpp
// 3. Подвох: это НЕ универсальная ссылка
template <typename T>
void vector_push(std::vector<T>&& v); // T выводится, но вид ссылки
                                      // искажен (vector<T>&&).
                                      // Это rvalue-ссылка на вектор.

// 4. Еще подвох: const убивает универсальность
template <typename T>
void const_wrapper(const T&& arg); // const T&& — это всегда rvalue-ссылка,
                                   // но только на const объекты.
```

Главное свойство универсальной ссылки: она "впитывает" категорию значения (value category) переданного аргумента.

Математика ссылок: Reference Collapsing

Как компилятор понимает, во что превратить T&&? Здесь вступает в силу правило **схлопывания ссылок** (Reference Collapsing).

В C++ запрещено создавать "ссылку на ссылку" явно (нельзя написать `int& & x`), но компилятор может сгенерировать такую конструкцию в процессе инстанцирования шаблона. Когда это происходит, две ссылки объединяются в одну по строгим правилам.

Пусть мы передаем аргумент в функцию `template <typename T> void f(T&& param)`.

1. **Сценарий 1: Передаем lvalue (например, int x).** Компилятор выводит T как `int&` (lvalue-ссылка). Подставляем в сигнатуру T&&:

   $$\texttt{int\&} + \&\& \rightarrow \texttt{int\&}$$

   Результат: Функция принимает lvalue-ссылку.

2. **Сценарий 2: Передаем rvalue (например, 42).** Компилятор выводит T как `int` (не ссылка). Подставляем в сигнатуру T&&:

   $$\texttt{int} + \&\& \rightarrow \texttt{int\&\&}$$

   Результат: Функция принимает rvalue-ссылку.

Полная таблица правил схлопывания (где левая часть — тип T, правая — спецификатор параметра):

Тип T	Спецификатор	Результат
type&	&	type&
type&	&&	type&
type&&	&	type&
type&&	&&	type&&

Таблица 2.1: Правила Reference Collapsing

> **Важно!**
>
> Интуитивное правило: **Lvalue-ссылка заразна**. Если в уравнении появляется хотя бы один одиночный амперсанд (&), результат всегда будет lvalue-ссылкой. Rvalue-ссылка получается только из комбинации двух двойных амперсандов (или типа без ссылок + &&).

Идеальная передача: std::forward vs std::move

Универсальные ссылки чаще всего используются для *Perfect Forwarding* — передачи аргументов в другую функцию с сохранением их категории (lvalue остается lvalue, rvalue остается rvalue).

Почему нельзя использовать std::move?

Рассмотрим, что произойдет, если мы применим `std::move` к универсальной ссылке.

```
void process(const Widget& lval) { std::cout << "Lvalue processed\n"; }
void process(Widget&& rval)      { std::cout << "Rvalue processed\n"; }

template <typename T>
void log_and_call(T&& arg) {
    std::cout << "Logging...\n";
    // ОШИБКА! std::move безусловно кастит к rvalue!
    process(std::move(arg));
}

int main() {
    Widget w;
    log_and_call(w); // Мы передаем lvalue...
}
```

В примере выше: 1. Вызывается `log_and_call(w)`. `T` выводится как `Widget&`. 2. `arg` имеет тип `Widget&`. 3. `std::move(arg)` кастит его к `Widget&&` (rvalue). 4. Вызывается перегрузка `process(Widget&&)`. 5. **Результат:** Мы передали в функцию живой lvalue объект `w`, но функция `process` считает его временным и может "украсть" его данные. После возврата из `log_and_call` объект `w` может оказаться пустым. Это катастрофа.

Решение: std::forward

Для решения этой проблемы используется `std::forward<T>`. В отличие от `std::move`, который делает *безусловный* каст, `std::forward` делает *условный* каст.

- Если `T` — lvalue-ссылка, `forward` возвращает lvalue.
- Если `T` — не ссылка (или rvalue-ссылка), `forward` возвращает rvalue.

```
template <typename T>
void log_and_call(T&& arg) {
    std::cout << "Logging...\n";
    // T передается явно как параметр шаблона forward
    process(std::forward<T>(arg));
}
```

Теперь при вызове `log_and_call(w)` (lvalue), `std::forward` вернет `Widget&`, и вызовется безопасная перегрузка. При вызове `log_and_call(Widget())` (rvalue), вернется `Widget&&`, и ресурсы будут эффективно перемещены.

Ref-qualifiers: Перегрузка методов для *this

Иногда нам нужно знать категорию значения не аргумента, а самого объекта, у которого вызывается метод (`*this`). Это особенно актуально для паттерна **Builder** или для оптимизации геттеров.

До C++11 мы могли перегружать методы только по `const`. C++11 добавил **Ref-qualifiers**: возможность указывать & или && после списка аргументов метода.

Пример: Паттерн Builder

Представим построитель сложного объекта (например, поискового индекса).

```
class IndexBuilder {
    std::vector<int> data;
public:
    void add(int x) { data.push_back(x); }

    // Возвращаем результат
    std::vector<int> finish() {
        return data; // Копирование! (или NRVO)
    }
};
```

Мы хотим, чтобы метод `finish()` перемещал данные, если `Builder` является временным объектом, и копировал, если долгоживущим.

```
class EfficientBuilder {
    std::vector<int> data;
public:
    void add(int x) { data.push_back(x); }

    // Версия для lvalue (долгоживущий объект): копируем
    std::vector<int> finish() & {
        return data;
    }

    // Версия для rvalue (временный объект): перемещаем
    std::vector<int> finish() && {
        return std::move(data);
    }
};

void usage() {
    EfficientBuilder b;
    b.add(1);

    // Вызывается finish() & -> копирование
    auto v1 = b.finish();

    // Вызывается finish() && -> перемещение
    // b становится пустым, но это безопасно, так как мы сделали move(b)
    auto v2 = std::move(b).finish();
```

```
27
28      // Строим и сразу забираем -> перемещение
29      auto v3 = EfficientBuilder().finish();
30  }
```

Это мощный механизм, но он имеет недостаток: дублирование кода. Если логика метода сложная, нам приходится писать две (или четыре, с учетом `const`) почти одинаковые функции.

C++23: Deducing This

Стандарт C++23 представил революционное изменение в синтаксисе методов классов, названное **Deducing This** (или *Explicit Object Parameter*).

Теперь мы можем явно объявить параметр, который будет представлять `this`, и сделать его шаблонным. Это позволяет "выводить" категорию значения самого объекта и использовать **Perfect Forwarding** для `*this`.

Синтаксис выглядит так: первым параметром метода идет `this Self&& self`.

Упрощение Ref-qualifiers

Перепишем пример с `EfficientBuilder` на C++23. Вместо двух функций нам понадобится одна.

```
1  #include <vector>
2  #include <utility> // для std::forward
3
4  class ModernBuilder {
5      std::vector<int> data;
6  public:
7      void add(int x) { data.push_back(x); }
8
9      // Единый шаблонный метод
10     // Self выведется как ModernBuilder& (для lvalue)
11     // или ModernBuilder (для rvalue)
12     template <typename Self>
13     auto finish(this Self&& self) {
14         // self — это наш *this, но теперь это обычный аргумент.
15         // Мы можем применить std::forward к самому объекту!
16
17         // forward_like - новая утилита C++23, аналогичная forward,
18         // но применяющая категорию self к полю data.
19         return std::forward<Self>(self).data;
20     }
21 };
```

На заметку

Как это работает:

1. Если мы зовем `b.finish()` (где `b` — lvalue), `Self` выводится как `ModernBuilder&`. `std::forward` возвращает ссылку, происходит копирование вектора.
2. Если мы зовем `std::move(b).finish()`, `Self` выводится как `ModernBuilder`. `std::forward` кастит `self` к rvalue. При доступе к `.data` мы получаем rvalue-доступ к полю, и срабатывает move-конструктор возвращаемого значения.

Преимущества Deducing This:

- **Дедупликация:** Одна реализация вместо 4-х (const/non-const, &/&&).
- **Рекурсивные лямбды:** Теперь лямбда может легко ссылаться на саму себя через явный параметр `this auto&& self`.
- **CRTP:** Паттерн Curiously Recurring Template Pattern становится проще, так как в базовом классе можно принимать `this Derived&& self` без приведения типов.

Резюме раздела

- `T&&` в шаблонах — это **Forwarding Reference**, а не rvalue-ссылка.
- `std::forward` обязателен при передаче универсальных ссылок дальше по стеку вызовов.
- Правила **Reference Collapsing** гарантируют, что lvalue превращает всю цепочку ссылок в lvalue.
- C++23 **Deducing This** позволяет писать обобщенный код, который автоматически адаптируется к `const` и `rvalue/lvalue` состоянию объекта.

Глава 3

Архитектура эксклюзивного владения: std::unique_ptr

Умные указатели — краеугольный камень современного C++. В этой главе мы не просто рассмотрим, "как пользоваться" `std::unique_ptr`, а разберем его устройство до байта. Мы реализуем собственный `unique_ptr`, столкнемся с проблемой пустого базового класса (EBO) и узнаем, как специализировать шаблоны для массивов.

Концепция эксклюзивного владения

`std::unique_ptr<T>` моделирует семантику единоличного владения ресурсом. Это означает:

1. В любой момент времени ресурсом владеет ровно один указатель.
2. Копирование указателя запрещено (компилятор выдаст ошибку).
3. Перемещение разрешено (владение переходит от A к B, A становится пустым).
4. При уничтожении указателя ресурс освобождается автоматически.

Это идеальная абстракция для ресурсов, которые не должны быть разделены: дескрипторы файлов, соединения с базой данных, динамические массивы внутри классов.

Базовый интерфейс

Минимальный интерфейс `unique_ptr` выглядит так:

```
template <typename T>
class UniquePtr {
    T* ptr_ = nullptr;

public:
    // Конструктор по умолчанию и от сырого указателя
    explicit UniquePtr(T* ptr = nullptr) : ptr_(ptr) {}

    // ЗАПРЕТ копирования (Rule of 5)
    UniquePtr(const UniquePtr&) = delete;
    UniquePtr& operator=(const UniquePtr&) = delete;
```

```
12
13      // Move-конструктор
14      UniquePtr(UniquePtr&& other) noexcept
15          : ptr_(std::exchange(other.ptr_, nullptr)) {}
16
17      // Move-оператор присваивания
18      UniquePtr& operator=(UniquePtr&& other) noexcept {
19          if (this != &other) {
20              delete ptr_; // Очищаем свой ресурс
21              ptr_ = std::exchange(other.ptr_, nullptr); // Забираем чужой
22          }
23          return *this;
24      }
25
26      // Деструктор
27      ~UniquePtr() {
28          delete ptr_;
29      }
30
31      // Доступ к данным
32      T& operator*() const { return *ptr_; }
33      T* operator->() const { return ptr_; }
34      T* get() const { return ptr_; }
35  };
```

Release vs Reset

Два важнейших метода управления владением:

- `reset(ptr)`: Уничтожает старый объект (вызывает `delete`) и захватывает новый `ptr`. Если вызван без аргументов, просто уничтожает объект.
- `release()`: Отказывается от владения, **не уничтожая** объект. Возвращает сырой указатель и зануляет внутреннее поле. Используется для передачи владения в C-API или легаси код.

```
1  void reset(T* ptr = nullptr) {
2      T* old_ptr = std::exchange(ptr_, ptr);
3      delete old_ptr; // Удаляем старый ресурс
4  }
5
6  T* release() {
7      return std::exchange(ptr_, nullptr); // Просто отдаем и забываем
8  }
```

Проблема размера (The Sizeof Problem)

В реальном `std::unique_ptr` есть второй шаблонный параметр — **Deleter**. Это функтор, который определяет, *как* удалять ресурс.

```cpp
template <typename T, typename Deleter = std::default_delete<T>>
class unique_ptr {
    T* ptr_;
    Deleter deleter_; // Храним экземпляр дилитера
public:
    ~unique_ptr() {
        if (ptr_) deleter_(ptr_); // Вызов функтора
    }
};
```

По умолчанию `Deleter` — это пустая структура:

```cpp
template <typename T>
struct default_delete {
    void operator()(T* ptr) const { delete ptr; }
};
```

Проблема:

1. `sizeof(T*)` = 8 байт (на 64-бит).
2. `sizeof(default_delete)` = 1 байт (пустая структура не может иметь размер 0, иначе у неё не будет адреса).
3. Из-за выравнивания (alignment) компилятор добавит padding.

Итог: `sizeof(unique_ptr)` становится $8 + 1 + \text{padding}(7) = 16$ байт. Это недопустимо! Умный указатель должен весить столько же, сколько сырой (Zero Overhead Principle).

Empty Base Optimization (EBO)

В C++ есть специальное правило: **базовый класс** может иметь размер 0 байт, если он пустой. Это называется *Empty Base Optimization*.

Чтобы воспользоваться этим, мы не можем хранить `Deleter` как поле. Мы должны от него **унаследоваться**. Но мы не можем просто унаследовать `unique_ptr` от `Deleter`, так как `Deleter` может быть:

- Финальным классом (final).
- Указателем на функцию (наследоваться нельзя).
- Ссылкой.

Решение: использование вспомогательной структуры `CompressedPair`.

Реализация CompressedPair

Идея: храним два значения. Если второе пустое — наследуемся от него.

```cpp
// Основной шаблон (если T2 не пустой)
template <typename T1, typename T2, bool = std::is_empty_v<T2>>
class CompressedPair {
```

```
    T1 first_;
    T2 second_;
public:
    T1& GetFirst() { return first_; }
    T2& GetSecond() { return second_; }
};

// Специализация для пустого T2 (включаем EBO)
template <typename T1, typename T2>
class CompressedPair<T1, T2, true> : private T2 { // Наследуемся!
    T1 first_;
public:
    T1& GetFirst() { return first_; }
    T2& GetSecond() { return *this; } // Кастим себя к базе
};
```

Теперь, используя `CompressedPair<T*, Deleter>` внутри `unique_ptr`, мы получаем размер 8 байт для stateless дилитеров.

Специализация для массивов

`std::unique_ptr` поддерживает массивы: `std::unique_ptr<int[]>`. Это критически важно, потому что для массивов нужно вызывать `delete[]` вместо `delete`.

Если бы специализации не было:

```
std::unique_ptr<int> p(new int[10]);
// Деструктор вызовет delete p (без скобок) -> UB!
```

Как реализована специализация `unique_ptr<T[]>`: 1. Дефолтный дилитер вызывает `delete[]`. 2. Оператор `*` и → удалены (логически к массиву нельзя применить →). 3. Добавлен оператор индексации `operator[]`.

```
template <typename T, typename Deleter>
class unique_ptr<T[], Deleter> {
public:
    // ...
    T& operator[](size_t i) const {
        return ptr_[i];
    }

    // operator* и operator-> отсутствуют
};
```

Угловой кейс: unique_ptr<void>

Можно ли создать `std::unique_ptr<void>`?

С дефолтным дилитером — **нет**.

```
std::unique_ptr<void> p(malloc(100)); // Ошибка компиляции или UB
```

Причина: `delete ptr`, где `ptr` имеет тип `void*`, является Undefined Behavior (если мы не знаем реальный тип объекта, мы не можем вызвать его деструктор). Стандартная библиотека часто ставит `static_assert(!is_void_v<T>)` в `default_delete`.

Однако, `unique_ptr<void>` полезен для управления памятью, выделенной через `malloc`, или для C-API хендлов. Для этого **обязательно** нужно предоставить кастомный дилитер.

```
struct FreeDeleter {
    void operator()(void* p) const {
        std::free(p); // free корректно работает с void*
    }
};

// Теперь это легально и безопасно
std::unique_ptr<void, FreeDeleter> memory(std::malloc(1024));
```

Это мощный паттерн для RAII-оберток над любыми ресурсами C-библиотек (FILE*, SDL_Surface*, SSL*).

Резюме раздела

- `unique_ptr` — это "золотой стандарт" владения. Нулевой оверхед.
- **EBO (Empty Base Optimization)** позволяет дилитеру не занимать память, если он не имеет состояния.
- Специализация для массивов `<T[]>` меняет `delete` на `delete[]` и добавляет `operator[]`.
- Для `void*` или системных ресурсов всегда используйте кастомный дилитер.

Глава 4

Внутреннее устройство разделяемого владения: std::shared_ptr

В отличие от `std::unique_ptr`, который обеспечивает эксклюзивное владение, `std::shared_ptr` реализует модель совместного (разделяемого) владения ресурсом. Несколько указателей могут ссылаться на один и тот же объект, и этот объект будет уничтожен только тогда, когда исчезнет *последний* указатель.

В этой главе мы погрузимся в архитектуру `shared_ptr`, разберем структуру контрольного блока, поймем, почему `sizeof(shared_ptr)` равен двум указателям, и научимся использовать "Aliasing Constructor" — одну из самых мощных, но малоизвестных возможностей C++.

Анатомия Shared Ptr: Два указателя

На первый взгляд может показаться, что `shared_ptr` просто хранит указатель на объект и счетчик ссылок. Но где хранится этот счетчик? Он не может быть внутри объекта (это требовало бы изменения класса объекта, что называется интрузивным подходом). Он не может быть статическим полем (тогда счетчик был бы общим для всех объектов типа T).

Решение: `shared_ptr` создает вспомогательный объект в куче, который называется **Control Block** (контрольный блок).

Сам объект `shared_ptr` на стеке состоит из двух полей:

1. `T* stored_ptr` — "сырой" указатель на управляемый объект. Используется для операторов разыменования (*, →).
2. `ControlBlock* cb_ptr` — указатель на контрольный блок.

Таким образом, на 64-битной архитектуре `sizeof(std::shared_ptr<T>) == 16 байт`.

Структура Control Block

Контрольный блок — это сердце механизма shared ownership. Он содержит:

- **Strong Ref Count** (счетчик сильных ссылок): количество живых `shared_ptr`.
- **Weak Ref Count** (счетчик слабых ссылок): количество живых `weak_ptr`.
- **Deleter**: функтор для уничтожения объекта (стирание типа через виртуальный вызов).

- **Allocator**: аллокатор, использованный для выделения памяти под контрольный блок.

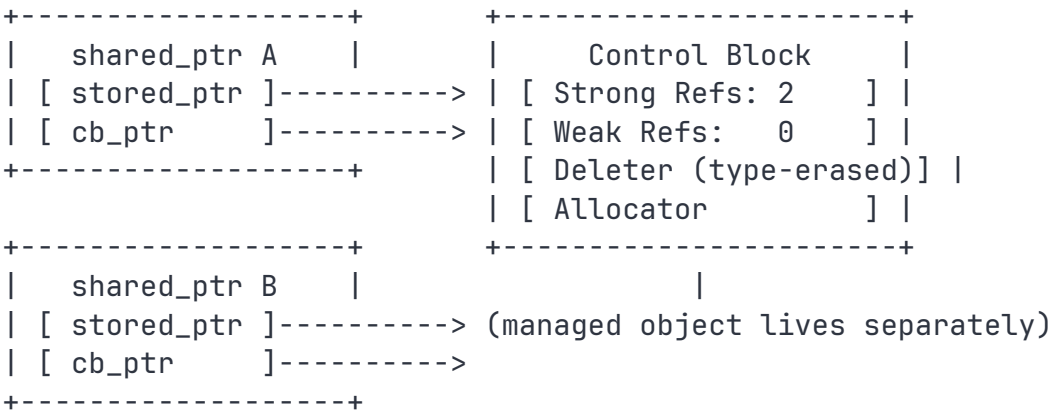


Рис. 4.1: Схема памяти обычного shared_ptr

std::make_shared vs std::shared_ptr(new T)

Существует два способа создать `shared_ptr`:

```
// 1. Через конструктор
std::shared_ptr<int> p1(new int(42));

// 2. Через make_shared
auto p2 = std::make_shared<int>(42);
```

Разница между ними фундаментальна с точки зрения производительности и размещения памяти.

Проблема двойной аллокации

При вызове `shared_ptr(new T)` происходят две независимые аллокации памяти: 1. `new int(42)` выделяет память под объект. 2. Конструктор `shared_ptr` выделяет память под **Control Block**.

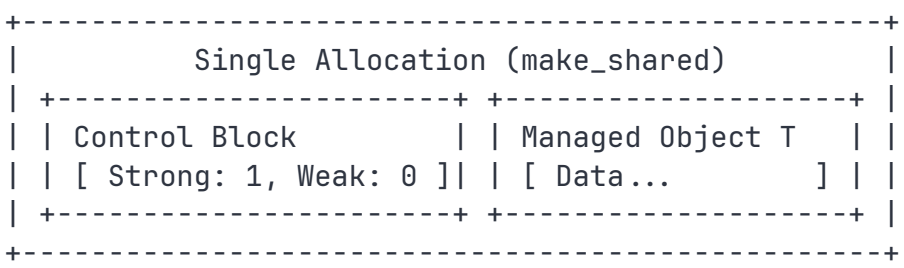
Это плохо по двум причинам:

- Лишняя нагрузка на аллокатор (медленно).
- Плохая локальность данных (объект и счетчик могут лежать далеко друг от друга, вызывая cache miss при доступе).
- **Небезопасность исключений:** Если мы пишем `f(shared_ptr<int>(new int(42)), g())`, и `g()` кидает исключение *после* `new int`, но *до* конструктора `shared_ptr`, мы получаем утечку памяти.

Оптимизация make_shared

`std::make_shared` совершает **одну большую аллокацию**, в которой размещает и Control Block, и сам объект T "паровозиком" (рядом друг с другом).

Это экономит память (меньше оверхеда на заголовки аллокатора) и процессорное время.



> **Важно!**
>
> У `make_shared` есть один неочевидный недостаток. Память под объект T не может быть освобождена, пока жив хотя бы один `weak_ptr`. Поскольку блок и объект — это единый кусок памяти, они живут и умирают вместе. Если у вас огромный объект и долгоживущие слабые ссылки, лучше использовать раздельные аллокации.

Aliasing Constructor (Конструктор псевдонимов)

Это "секретное оружие" `shared_ptr`. Сигнатура конструктора выглядит так:

```
template <typename Y>
shared_ptr(const shared_ptr<Y>& r, T* ptr) noexcept;
```

Что он делает:

- Он берет **Control Block** (владение) от указателя `r`. То есть он увеличивает счетчик ссылок того объекта, которым владеет `r`.
- Но в качестве `stored_ptr` (то, что возвращает `operator*`) он запоминает `ptr`.

Суть: Мы владеем одним объектом (и продлеваем ему жизнь), но указываем на другой (обычно — на его часть).

Пример: Указатель на поле структуры

Представьте, что у нас есть структура `Message`, и мы хотим передать в функцию только одну её строку `payload`, но так, чтобы вся структура `Message` не была удалена, пока мы работаем со строкой.

```
struct Message {
    Header header;
    std::string payload; // Хотим указывать сюда
};

void ProcessPayload(std::shared_ptr<std::string> str_ptr) {
    std::cout << *str_ptr << "\n";
}

int main() {
    // Создаем сообщение (владеем всем объектом)
    auto msg = std::make_shared<Message>();
    msg->payload = "Hello World";
```

```
14
15      // Создаем алиас-указатель
16      // 1. Владеем msg (увеличиваем refcount для Message)
17      // 2. Указываем на msg->payload
18      std::shared_ptr<std::string> payload_ptr(msg, &msg->payload);
19
20      // Теперь msg можно "забыть" в этой scope
21      msg.reset();
22
23      // Но объект Message НЕ будет удален!
24      // Потому что payload_ptr держит Control Block от Message.
25      ProcessPayload(payload_ptr);
26
27      // Message удалится только здесь, когда умрет payload_ptr
28  }
```

Если бы мы просто создали `shared_ptr<string>(&msg→payload)`, это привело бы к катастрофе: новый `shared_ptr` попытался бы сделать `delete` для поля внутри структуры, что является некорректным (freeing stack/middle memory). Aliasing constructor решает эту проблему, разделяя понятия "чем владеем" и "на что указываем".

Жизненный цикл Control Block

Механизм удаления в `shared_ptr` работает в два этапа, и это важно понимать при реализации.

1. **Уничтожение Объекта (Object Destruction):** Когда `Strong Ref Count` достигает 0:
   - Вызывается `Deleter(object_ptr)`.
   - Вызывается деструктор объекта T.
   - Память под объект освобождается (если это не `make_shared`).

   На этом этапе `weak_ptr` переходят в состояние *expired*.
2. **Уничтожение Контрольного блока (Block Deallocation):** Когда `Strong Ref Count` == 0 **И** `Weak Ref Count` == 0:
   - Удаляется сам Control Block.
   - Если использовался `make_shared`, только в этот момент освобождается вся память (и блока, и объекта).

Именно поэтому `weak_ptr` удерживает память контрольного блока (а в случае `make_shared` — и память мертвого объекта) от физического освобождения.

Реализация логики деструктора

В упрощенном виде логика декремента счетчика в деструкторе `shared_ptr` выглядит так:

```
1  ~shared_ptr() {
2      if (control_block) {
3          // Атомарный декремент
```

```
        if (control_block->strong_refs.fetch_sub(1) == 1) {
            // Мы были последним владельцем
            control_block->dispose_object(); // Уничтожить T

            // Проверяем слабые ссылки
            if (control_block->weak_refs.load() == 0) {
                control_block->destroy_self(); // Уничтожить блок
            }
        }
        // Важно: декремент слабых ссылок происходит в ~weak_ptr
        // и там тоже есть проверка на удаление блока.
    }
}
```

На самом деле, логика немного сложнее: сильный счетчик тоже неявно считается "одной слабой ссылкой", чтобы блок жил, пока жив хоть один владелец. Обычно используется схема: "Block живет, пока `Weak + (Strong > 0 ? 1 : 0) > 0`".

Резюме раздела

- `shared_ptr` состоит из двух указателей: на объект и на Control Block.
- `make_shared` эффективнее (1 аллокация), но может удерживать память дольше из-за слабых ссылок.
- **Aliasing Constructor** позволяет создавать `shared_ptr`, которые владеют объектом A, но указывают на объект B.
- Контрольный блок живет дольше управляемого объекта, если есть `weak_ptr`.

Глава 5

Безопасное управление жизненным циклом: WeakPtr и ESFT

В этой главе мы разберем две взаимосвязанные концепции, без которых экосистема `shared_ptr` была бы неполной и опасной: `std::weak_ptr` и идиому `enable_shared_from_this`. Мы увидим, как разрывать циклические ссылки, почему нельзя создавать умные указатели из `this` напрямую, и как C++ решает эту проблему через сложную машинерию SFINAE и наследования.

Проклятие циклических ссылок

Модель подсчета ссылок (*Reference Counting*) имеет фундаментальный недостаток: она не умеет обрабатывать циклы.

Представьте два объекта, которые ссылаются друг на друга через `shared_ptr`:

```
struct Node {
    std::shared_ptr<Node> neighbor;
    ~Node() { std::cout << "Deleted\n"; }
};

void create_cycle() {
    auto a = std::make_shared<Node>(); // A refs=1
    auto b = std::make_shared<Node>(); // B refs=1

    a->neighbor = b; // B refs=2
    b->neighbor = a; // A refs=2
}
// При выходе из функции:
// a уничтожается -> A refs=1
// b уничтожается -> B refs=1
// Итог: оба объекта живы, память утекла навсегда. Деструкторы не вызваны.
```

Чтобы разорвать цикл, одна из ссылок должна быть "слабой" — то есть не влиять на время жизни объекта, но позволять проверить, жив ли он.

std::weak_ptr: Наблюдатель

`std::weak_ptr` — это умный указатель, который ссылается на объект, управляемый `shared_ptr`, но не увеличивает *Strong Ref Count*. Вместо этого он увеличивает *Weak Ref Count* в контрольном блоке.

Свойства `weak_ptr`:

- Его нельзя разыменовать напрямую (операторов `*` и `->` нет). Это сделано специально: объект может исчезнуть в любой момент (в другом потоке), поэтому доступ к нему должен быть транзакционным.
- Метод `expired()` проверяет, удален ли объект.
- Метод `lock()` пытается создать `shared_ptr` из слабой ссылки. Если объект жив, мы получаем валидный `shared_ptr` (и счетчик сильных ссылок временно растет). Если мертв — получаем `nullptr` (пустой `shared_ptr`).

Исправление примера с циклом:

```
struct Node {
    // Используем weak_ptr для обратной ссылки
    std::weak_ptr<Node> neighbor;
};

void safe_cycle() {
    auto a = std::make_shared<Node>();
    auto b = std::make_shared<Node>();

    a->neighbor = b; // std::shared_ptr конвертируется в weak_ptr
    b->neighbor = a;
}
// При выходе:
// a уничтожается -> A refs=0 (удаляется!)
// Деструктор A уничтожает поле neighbor (слабую ссылку на B).
// b уничтожается -> B refs=0 (удаляется!).
// Память чиста.
```

Проблема "this" и enable_shared_from_this

Часто возникает ситуация, когда объект внутри своего метода хочет передать указатель на самого себя в функцию, принимающую `shared_ptr`.

```
struct Widget {
    void register_in_system(std::vector<std::shared_ptr<Widget>>& registry) {
        // ОШИБКА! Создается НОВЫЙ контрольный блок!
        registry.emplace_back(this);
    }
};

int main() {
    auto w = std::make_shared<Widget>(); // Control Block 1 (refs=1)

```

```
    std::vector<std::shared_ptr<Widget>> reg;
    w->register_in_system(reg); // Создает Control Block 2 (refs=1)

    // При выходе:
    // w уничтожается -> удаляет Widget (Block 1)
    // reg уничтожается -> удаляет ТОТ ЖЕ Widget (Block 2)
    // Double Free! Крэш программы.
}
```

Проблема в том, что сырой указатель `this` ничего не знает о том, что он уже управляется каким-то `shared_ptr` где-то снаружи. Создавая `shared_ptr(this)`, мы создаем *вторую независимую иерархию владения* для того же адреса памяти.

Решение: std::enable_shared_from_this

Стандартная библиотека предлагает паттерн (Mix-in класс) `std::enable_shared_from_this<T>`.

Механика работы "под капотом": 1. Класс `Widget` наследуется от `std::enable_shared_from_this<W` 2. В базовом классе хранится скрытое поле `std::weak_ptr<Widget> weak_this`. 3. Когда мы создаем `shared_ptr<Widget>`, конструктор `shared_ptr` проверяет (через SFINAE или концепты), является ли `Widget` наследником `ESFT`. 4. Если да, то `shared_ptr` инициализирует поле `weak_this` внутри объекта, записывая туда слабую ссылку на *тот самый* контрольный блок, который он только что создал.

Теперь объект может восстановить `shared_ptr` на себя, используя сохраненный `weak_this`.

```
// 1. Наследуемся (CRTP паттерн)
struct Widget : public std::enable_shared_from_this<Widget> {

    void register_in_system(std::vector<std::shared_ptr<Widget>>& registry) {
        // 2. Используем shared_from_this()
        registry.push_back(shared_from_this());
    }
};

int main() {
    auto w = std::make_shared<Widget>();
    // Внутри make_shared происходит магия инициализации weak_this

    std::vector<std::shared_ptr<Widget>> reg;
    w->register_in_system(reg); // Всё безопасно, счетчик увеличился до 2
}
```

Ловушки ESFT

Несмотря на удобство, `enable_shared_from_this` имеет строгие ограничения.

Ловушка 1: Вызов в конструкторе

> **Важно!**
>
> Нельзя вызывать `shared_from_this()` в конструкторе объекта!

Почему? В момент работы конструктора `shared_ptr`, который будет владеть объектом, **еще не завершил инициализацию**. Контрольный блок, возможно, уже есть, но инициализация поля `weak_this` происходит *после* завершения конструктора объекта (обычно в конструкторе `shared_ptr`).

Если вызвать `shared_from_this()` в конструкторе, поле `weak_this` еще пустое (expired), и будет выброшено исключение `std::bad_weak_ptr`.

Ловушка 2: Объект на стеке

Если вы создадите объект на стеке:

```
Widget w;
w.shared_from_this(); // Крэш (bad_weak_ptr)
```

У стекового объекта нет контрольного блока, `weak_this` не инициализирован. `ESFT` работает **только** если объект управляется через `shared_ptr`.

Реализация инициализации weak_this (Deep Dive)

Как именно `shared_ptr` узнает, что нужно инициализировать поле внутри T? Это делается через шаблонную магию. Упрощенно:

```
template<typename T>
class shared_ptr {
public:
    template<typename Y>
    shared_ptr(Y* ptr) {
        // ... создание control block ...

        // Магия: если Y наследуется от enable_shared_from_this,
        // вызвать приватный метод инициализации.
        if constexpr (std::is_base_of_v<std::enable_shared_from_this<Y>, Y>) {
            ptr->weak_this = *this; // присваиваем shared_ptr в weak_ptr
        }
    }
};
```

В реальности поле `weak_this` приватное, и `shared_ptr` объявлен другом для `enable_shared_from_t` чтобы иметь к нему доступ.

Резюме раздела

- `weak_ptr` — единственный способ разорвать циклические ссылки при использовании `shared_ptr`.
- Никогда не делайте `shared_ptr(this)`. Это приведет к Double Free.
- Используйте `std::enable_shared_from_this` для безопасного получения `shared_ptr` на себя.
- Помните, что `shared_from_this()` не работает в конструкторах и для объектов на стеке.

Глава 6

Стирание типов и Интрузивные указатели

В завершающей главе этого раздела мы рассмотрим продвинутые паттерны управления памятью и типами, которые выходят за рамки стандартных `unique_ptr` и `shared_ptr`. Мы реализуем собственный аналог `std::any` с использованием техники Type Erasure (стирание типов) и разберем концепцию интрузивных указателей — высокопроизводительной альтернативы `shared_ptr`, используемой в ядрах ОС, игровых движках и LLVM.

Паттерн Type Erasure: Реализация std::any

C++ — язык со строгой статической типизацией. Однако иногда нам нужен полиморфизм без наследования: возможность положить в контейнер объект *любого* типа (int, string, UserClass) и корректно управлять его временем жизни. Стандартный класс `std::any` делает именно это.

Как он работает внутри? Как он может хранить `T`, не будучи шаблонным классом `any<T>`? Ответ кроется в идиоме Type Erasure.

Архитектура Any

Суть идиомы:

1. Класс-обертка (`Any`) хранит указатель на *нешаблонный* абстрактный базовый класс (например, `Base`).
2. Внутри методов (обычно в конструкторе) мы создаем *шаблонного* наследника (`Derived<T>`), который знает конкретный тип `T`.
3. Указатель на наследника сохраняется в поле типа `Base*`.
4. Виртуальные функции в `Base` (в первую очередь деструктор) позволяют манипулировать объектом, не зная его реального типа в месте хранения.

Реализация

```
class Any {
private:
    // 1. Абстрактный интерфейс (Type Erased)
    struct Base {
```

```
        virtual ~Base() = default;
        virtual std::unique_ptr<Base> clone() const = 0;
    };

    // 2. Шаблонная реализация (Type Aware)
    template <typename T>
    struct Derived : Base {
        T value;

        Derived(const T& v) : value(v) {}
        Derived(T&& v) : value(std::move(v)) {}

        std::unique_ptr<Base> clone() const override {
            return std::make_unique<Derived<T>>(value);
        }
    };

    std::unique_ptr<Base> storage;

public:
    // Конструктор: принимает любой тип
    template <typename T>
    Any(T&& value) {
        // Выводим чистый тип (без const/volatile/reference)
        using CleanT = std::decay_t<T>;

        // Создаем конкретного наследника, но храним как Base
        storage = std::make_unique<Derived<CleanT>>(std::forward<T>(value));
    }

    // Copy Constructor (использует clone)
    Any(const Any& other) {
        if (other.storage) {
            storage = other.storage->clone();
        }
    }

    // Вспомогательный метод для any_cast (небезопасный для простоты)
    template <typename T>
    T* cast() {
        if (auto d = dynamic_cast<Derived<T>*>(storage.get())) {
            return &d->value;
        }
        return nullptr;
    }
};
```

На заметку

Обратите внимание: класс `Any` **не является** шаблонным. Вы можете объявить `std::vector<Any>`, и в нем будут лежать объекты разных типов. Магия происходит в конструкторе: именно там "стирается" тип `T`, превращаясь в полиморфный указатель `Base*`.

Деструктор `Any` автоматически вызовет деструктор `unique_ptr<Base>`, который вызовет

`virtual Base()`, который диспетчеризируется в `Derived<T>()`, корректно уничтожая поле `value` конкретного типа.

Интрузивные указатели (IntrusivePtr)

`std::shared_ptr` удобен, но имеет недостатки:

- **Размер:** 16 байт (2 указателя).
- **Аллокация:** Требует внешней памяти под Control Block (даже с `make_shared` это оверхед на структуру блока).
- **Оторванность:** Нельзя просто так взять сырой указатель на объект и превратить его в `shared_ptr` (нужен `enable_shared_from_this` или новый блок).

Интрузивные указатели решают эти проблемы, требуя от объекта *самому* хранить счетчик ссылок.

Концепция

Объект класса `T` должен содержать поле `int ref_count`. Обычно это делается через наследование от базового класса `RefCounted`.

Умный указатель `IntrusivePtr<T>` хранит **только** сырой указатель `T*` (размер 8 байт). При копировании он вызывает метод `obj→add_ref()`, при уничтожении — `obj→release()`.

Реализация RefCounted

```
class RefCounted {
    mutable std::atomic<int> ref_count_ = 0;

protected:
    // Деструктор виртуальный, чтобы корректно удалять наследников
    virtual ~RefCounted() = default;

public:
    void add_ref() const {
        ref_count_.fetch_add(1, std::memory_order_relaxed);
    }

    void release() const {
        if (ref_count_.fetch_sub(1, std::memory_order_acq_rel) == 1) {
            delete this; // Объект удаляет сам себя!
        }
    }
};
```

> **Важно!**
>
> Конструкция `delete this` абсолютно легальна в C++, если после неё не обращаться к полям объекта. Именно так работают COM-объекты в Windows и многие структуры ядра Linux.

Сравнение с shared_ptr

Характеристика	std::shared_ptr	IntrusivePtr
Размер	16 байт (2 ptr)	8 байт (1 ptr)
Аллокации	1 (make_shared) или 2	0 (дополнительных)
Локальность	Хорошая (make_shared)	Идеальная (счетчик внутри)
Восстановление	Опасно (без ESFT)	Безопасно (счетчик всегда с собой)
Weak Ptr	Есть	Нет (обычно)
Требования к T	Любой тип	Должен наследовать RefCounted

Пример использования

```
class Texture : public RefCounted {
    // ... данные текстуры ...
};

void use_texture(IntrusivePtr<Texture> tex) {
    // Копирование tex увеличивает счетчик внутри самого объекта Texture
}

int main() {
    // Обычный new, никаких make_shared
    IntrusivePtr<Texture> t(new Texture());

    // Можно восстановить умный указатель из сырого
    Texture* raw = t.get();
    IntrusivePtr<Texture> t2(raw); // OK! Счетчик инкрементируется.
    // Оба указателя смотрят на один объект, счетчик = 2.
}
```

Итоги раздела Smart Pointers

Мы прошли путь от базового `unique_ptr` до написания собственной системы типов и интрузивного управления памятью.

Резюме раздела

- **Type Erasure** позволяет создавать контейнеры для разнородных типов (`std::any`, `std::function`), пряча шаблонную логику за полиморфным фасадом.
- **IntrusivePtr** — выбор для высоконагруженных систем, где критичен каждый байт и такт процессора, и где вы контролируете иерархию классов.
- Понимание устройства Control Block и EBO делает вас не просто пользователем C++, а инженером, понимающим цену каждой абстракции.