

HSE

Faculty of Computer Science

Конспект углубленно- го курса по C++

AUTHOR

Your Name

COURSE

Advanced C++
Fall 2023

Оглавление

1 Лекция 02 – Динамическая память. Move семантика	3
1 Фундаментальное управление памятью и RAII	4
1.1 1.1 Анатомия памяти процесса	4
1.2 1.2 Проблемы ручного управления памятью (C-style)	5
1.3 Утечки памяти и обработка ошибок	5
1.4 1.3 Идиома RAII (Resource Acquisition Is Initialization)	5
1.5 1.4 Механика new и delete	6
1.6 Undefined Behavior при смещивании new[] и delete	7
1.7 1.5 Умные указатели: std::unique_ptr	7
1.8 1.6 std::make_unique vs new	8
2 Система типов: Категории значений (Value Categories)	10
2.1 2.1 Таксономия значений (C++17)	10
2.2 2.2 Практический анализ категорий	11
2.3 Почему строковый литерал – это lvalue?	11
2.4 2.3 Rvalue-ссылки (T&&)	11
2.5 2.4 Materialization (C++17)	12
3 Семантика перемещения (Move Semantics): Механика и История	14
3.1 3.1 Исторический контекст: Проблема лишних копий	14
3.2 Трюк со swap	15
3.3 3.2 Катастрофа std::auto_ptr	15
3.4 3.3 Философия Move Semantics	15
3.5 Анатомия Move Constructor	16
3.6 Анатомия Move Assignment Operator	16
3.7 3.4 std::move – это ложь	17
3.8 3.5 Ловушка реализации: Именованные Rvalue-ссылки	17
3.9 3.6 Правило пяти (Rule of 5)	18
4 Глава 4. Безопасность исключений и Контейнеры	20
4.1 4.1 Проблема реаллокации вектора	20
4.2 4.2 Ключевое слово noexcept	21
4.3 4.3 std::move_if_noexcept	22
4.4 Идиома "Move or Copy"	22
5 Время жизни объектов (Object Lifetime) и Оптимизации	24
5.1 5.1 Temporary Lifetime Extension	24
5.2 5.2 Опасные паттерны и висячие ссылки (Dangling Refs)	25
5.3 Ловушка 1: Возврат ссылки на временный объект	25
5.4 Ловушка 2: Доступ к полю временного объекта	25
5.5 5.3 RVO и Copy Elision	26
5.6 5.4 Pessimizing Move	26
6 Perfect Forwarding и Универсальные ссылки	28
6.1 6.1 Проблема передачи аргументов	28

6.5	6.5 Практическое применение	30
6.6	1. Emplace-методы контейнеров	30
6.7	2. make_unique / make_shared	31
7	Продвинутые идиомы C++ и Архитектурные паттерны	32
7.1	7.1 Ref-qualifiers (Квалификаторы ссылок для методов)	32
7.2	7.2 Empty Base Optimization (EBO)	33
7.3	7.3 Опасность const T&&	34
7.4	7.4 Destructive Move vs Non-destructive Move	34

Часть I

Лекция 02 – Динамическая память. Move семантика

Глава 1

Фундаментальное управление памятью и RAII

Управление памятью является краеугольным камнем системного программирования на C++. Понимание того, где и как хранятся объекты, определяет не только производительность приложения, но и его корректность. Данная глава посвящена анатомии памяти процесса, проблемам ручного управления ресурсами, идиоме RAII и современным механизмам эксклюзивного владения.

1.1 Анатомия памяти процесса

В контексте C++ модель памяти процесса операционной системы традиционно разделяется на четыре основных сегмента. Каждый из них имеет свои характеристики времени жизни объектов и накладных расходов на доступ.

- **Стек (Stack, Automatic Storage Duration):** Область памяти, работающая по принципу LIFO (Last In, First Out). Выделение памяти сводится к изменению значения регистра указателя стека (Stack Pointer), что делает операции аллокации и деаллокации чрезвычайно быстрыми (наносекунды). Размер стека фиксирован и невелик (обычно 2–8 МБ), что делает его непригодным для хранения больших структур данных.
- **Куча (Heap, Dynamic Storage Duration):** Область памяти произвольного доступа. Управление осуществляется через системные вызовы (например, `malloc/new`). Аллокация требует поиска свободного блока памяти подходящего размера, что значительно медленнее стековых операций и может приводить к фрагментации памяти. Время жизни объектов здесь полностью контролируется программистом.
- **Статическая память (Static Storage Duration):** Предназначена для глобальных переменных и переменных, объявленных с модификатором `static`. Память выделяется при запуске программы и освобождается при ее завершении. Объекты инициализируются один раз.
- **Thread-Local Storage (TLS):** Специфическая область памяти, уникальная для каждого потока выполнения. Переменные с модификатором `thread_local` существуют в течение времени жизни потока.

1.2 Проблемы ручного управления памятью (C-style)

В языке С и в "legacy" C++ коде управление динамической памятью осуществлялось вручную через пары функций `malloc/free` или операторов `new/delete`. Этот подход порождает класс критических ошибок, связанных с человеческим фактором.

Утечки памяти и обработка ошибок

При возникновении исключительной ситуации или преждевременном возврате из функции программист обязан гарантировать вызов функции освобождения ресурсов. В коде с множественными точками выхода это приводит к дублированию логики очистки или использованию запутанных конструкций `goto`.

Рассмотрим пример обработки ресурсов в стиле С:

```

1 void process_data() {
2     int* buffer = (int*)malloc(1024 * sizeof(int));
3     if (!buffer) return;
4
5     if (!init_network()) {
6         free(buffer); // Ручная очистка при ошибке
7         return;
8     }
9
10    if (!compute_hash(buffer)) {
11        close_network(); // Необходимо закрыть предыдущий ресурс
12        free(buffer); // И освободить память
13        return;
14    }
15
16    // Основная логика
17    free(buffer);
18    close_network();
19 }
```

Любое изменение логики (добавление нового ресурса или условия) требует модификации всех веток обработки ошибок, что экспоненциально увеличивает риск утечки ресурса (resource leak) или двойного освобождения (double free).

1.3 Идиома RAI (Resource Acquisition Is Initialization)

RAII – фундаментальная идиома C++, связывающая время жизни ресурса (памяти, файлового дескриптора, мьютекса) с временем жизни объекта на стеке.

RAII

Захват ресурса происходит в конструкторе объекта, а освобождение – в его деструкторе. Поскольку деструкторы автоматических переменных гарантированно вызываются при выходе из области видимости (в том числе при раскрутке стека из-за исключения), утечки ресурсов становятся невозможными при корректном использовании.

Пример RAII-обертки для файлового дескриптора C-style:

```

1 #include <cstdio>
2 #include <stdexcept>
3
4 class FileHandle {
5     FILE* file_;
6
7 public:
8     explicit FileHandle(const char* filename) {
9         file_ = std::fopen(filename, "r");
10        if (!file_) {
11            throw std::runtime_error("Failed to open file");
12        }
13    }
14
15 ~FileHandle() {
16     if (file_) {
17         std::fclose(file_); // Гарантированное закрытие
18     }
19 }
20
21 // Запрет копирования для избежания double-close
22 FileHandle(const FileHandle&) = delete;
23 FileHandle& operator=(const FileHandle&) = delete;
24
25 // Метод доступа
26 FILE* get() const { return file_; }
27 };
28
29 void safe_processing() {
30     FileHandle fh("data.txt");
31     // ... работа с файлом ...
32     throw std::runtime_error("Error");
33     // Деструктор fh вызывается автоматически, файл закрывается.
34 }
```

Механизм размотки стека (stack unwinding) гарантирует вызов деструкторов для всех полностью сконструированных объектов в блоке до передачи управления обработчику исключения.

1.4 Механика new и delete

В C++ операторы new и delete выполняют две задачи: управление памятью и управление временем жизни объекта.

- **new expression:** 1. Вызывает функцию operator new для выделения "сырой" памяти (аналог malloc). 2. Конструирует объект в этой памяти (вызывает конструктор). 3. Возвращает типизированный указатель.
- **delete expression:** 1. Вызывает деструктор объекта. 2. Вызывает функцию operator delete для освобождения памяти (аналог free).

Важно!

C++ требует строгого соответствия форм аллокации и dealлокации:

- `new T → delete ptr`
- `new T[] → delete[] ptr`

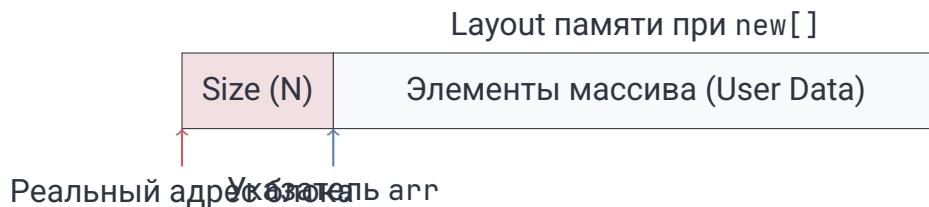
Смешивание этих форм (например, `delete` для массива) является Undefined Behavior.

Undefined Behavior при смешивании `new[]` и `delete`

Рассмотрим детально, почему следующий код вызывает неопределенное поведение, часто приводящее к повреждению кучи (heap corruption).

```
1 int* arr = new int[10];
2 delete arr; // ОШИБКА! UB
```

При использовании `new[]`, компилятор и аллокатор должны сохранить информацию о количестве элементов массива, чтобы `delete[]` мог вызывать деструкторы для **каждого** элемента. Часто это реализуется путем записи размера массива в памяти непосредственно перед адресом, возвращаемым пользователю (так называемый "cookie" или "overhead").



Сценарий сбоя:

1. `new int[10]` выделяет блок памяти размером $10 \times \text{sizeof(int)} + \text{sizeof(size_t)}$. Размер (10) записывается в начало, указатель сдвигается и возвращается программе.
2. `delete arr` (без скобок) предполагает, что удаляется **один** объект.
3. Если тип тривиальный (как `int`), деструкторы не вызываются, но `operator delete` получает адрес `arr`. Аллокатор ожидает, что ему передадут **реальный** адрес начала блока (который может быть сдвинут на размер cookie).
4. Передача неверного адреса в функцию освобождения памяти ломает внутренние структуры аллокатора (free list, метаданные страниц), что приводит к падению программы не сразу, а при следующей аллокации (delayed crash).

Для нетривиальных типов последствия еще хуже: вызывается деструктор только первого элемента, а остальные 9 объектов остаются несконструированными "зомби" или утекают, если они владели ресурсами.

1.5 Умные указатели: `std::unique_ptr`

Стандарт C++11 ввел `std::unique_ptr` как замену устаревшему `std::auto_ptr` и сырьем указателям. Это шаблонный класс, реализующий семантику эксклюзивного владения (exclusive ownership).

`std::unique_ptr<T>`

Обертка над сырым указателем, которая:

- Гарантирует вызов `delete` (или `delete[]`) в деструкторе.
- Не имеет оверхеда по памяти (размер равен `sizeof(void*)`, если используется дефолтный удалитель).
- Запрещает копирование (конструктор копирования и оператор присваивания удалены).
- Разрешает перемещение (Move Semantics), передавая владение другому объекту.

Специализация `std::unique_ptr<T[]>` корректно обрабатывает массивы, автоматически вызывая `delete[]`.

```

1 // Безопасное владение массивом
2 std::unique_ptr<int[]> safe_arr(new int[10]);
3 // При выходе из scope вызовется delete[]

```

1.6 `std::make_unique` vs `new`

Начиная с C++14, рекомендуется использовать фабричную функцию `std::make_unique` вместо прямого использования оператора `new`.

Проблема безопасности исключений:

Рассмотрим вызов функции, принимающей два уникальных указателя:

```

1 void process(std::unique_ptr<A> a, std::unique_ptr<B> b);
2
3 // ОПАСНЫЙ ВЫЗОВ
4 process(std::unique_ptr<A>(new A()), std::unique_ptr<B>(new B()));

```

C++ не гарантирует порядок вычисления аргументов функции. Возможна следующая последовательность операций компилятором:

1. `new A()` → выделена память для A.
2. `new B()` → **Выброшено исключение** (например, `std::bad_alloc`).
3. Конструктор `std::unique_ptr<A>` еще не был вызван.
4. Указатель на A теряется, происходит утечка памяти.

`std::make_unique` решает эту проблему, так как внутри функции создание объекта и обворачивание в умный указатель – это одна неразрывная операция.

```

1 // БЕЗОПАСНЫЙ ВЫЗОВ
2 process(std::make_unique<A>(), std::make_unique<B>());

```

Кроме безопасности, `make_unique` избавляет от дублирования типа (`unique_ptr<T>(new T)`) и делает код чище (принцип DRY).

Резюме раздела

- Используйте RAII для всех ресурсов.
- Избегайте сырых new/delete. Используйте std::unique_ptr и std::make_unique.
- Никогда не смешивайте скалярные и векторные формы new/delete.
- std::unique_ptr – это "бесплатная" абстракция с точки зрения производительности, обеспечивающая строгую семантику владения.

Глава 2

Система типов: Категории значений (Value Categories)

Понимание категорий значений – это необходимый фундамент для освоения семантики перемещения (Move Semantics) и идеальной передачи (Perfect Forwarding). В C++ каждое выражение (expression) характеризуется двумя независимыми свойствами: **типом** (например, `int`, `std::string`) и **категорией значения**.

Если тип определяет операции и представление в памяти, то категория значения определяет время жизни результата выражения и возможность его использования в качестве операнда для перемещения.

2.1 Таксономия значений (C++17)

До стандарта C++11 мир был прост: существовали только `lvalue` (left-hand side) и `rvalue` (right-hand side). С появлением семантики перемещения этой классификации стало недостаточно. Стандарт C++11 (и уточненный C++17) ввел более гранулярную систему.

В основе классификации лежат два ортогональных свойства выражения:

1. **Identity (m)**: Обладает ли выражение идентичностью? Иными словами, есть ли у результата выражения стабильный адрес в памяти, по которому можно к нему обратиться?
2. **Movability (m)**: Можно ли безопасно переместить ресурсы из объекта, который является результатом выражения? (Допускается ли "кража" состояния?)

На пересечении этих свойств образуются три фундаментальные (листовые) категории:

- **lvalue (identity + !movable)**: Именованные объекты, функции, поля данных. У них есть адрес, но перемещать из них неявно нельзя (чтобы не сломать логику программы).
- **rvalue (pure rvalue, !identity + movable)**: "Чистые" значения. Временные объекты, литералы (кроме строковых), результаты функций, возвращающих значение. Не имеют имени, существуют эфемерно.
- **xvalue (eXpiring value, identity + movable)**: Объекты, у которых есть адрес ("плоть"), но которые помечены как "умирающие". Программа явно разрешила перемещение из них (например, результат `std::move`).

Для удобства стандарт вводит две группирующие категории:

- **glvalue (generalized lvalue)**: Всё, что имеет идентичность ($\text{lvalue} \cup \text{xvalue}$).

- **rvalue:** Всё, из чего можно перемещать ($xvalue \cup prvalue$).

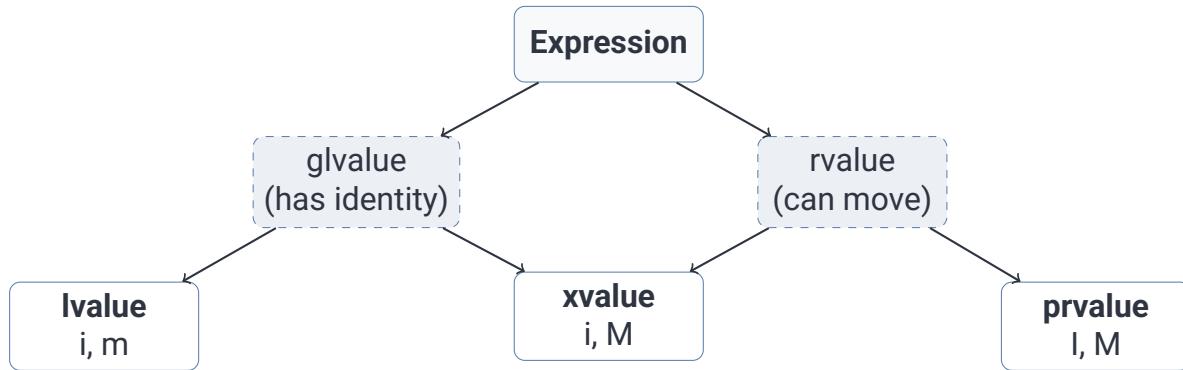


Рис. 2.1: Иерархия категорий значений в C++17

2.2 Практический анализ категорий

Рассмотрим код и определим категории значений для различных выражений. Ключевой маркер **lvalue** — возможность взять адрес через унарный оператор &.

```

1 int i = 1;                                // OK: i - это lvalue
2 int* y = &i;                                // ОШИБКА: 42 - это prvalue, адреса нет
3 // y = &42;                                // ОШИБКА: 1 - prvalue, нельзя присвоить в него
4 // 1 = i;
5
6 const char(*ptr)[5] = &"abcd"; // ВНИМАНИЕ: "abcd" - это lvalue!
7 char arr[20];
8 arr[10] = 'z';                            // Результат arr[10] - lvalue (ссылка на char)
9
10 int a, b;
11 (a, b) = 5;                                // Оператор запятая возвращает lvalue (ссылку на b)

```

Почему строковый литерал — это lvalue?

Это классический вопрос на понимание. Литерал 42 или 3.14 является **prvalue** — это просто битовое представление значения, которое может быть "зашито" прямо в инструкции процессора (immediate operand). У него нет адреса в оперативной памяти с точки зрения абстрактной машины C++.

Строчный литерал "abcd", напротив, имеет тип **const char[5]**. Массивы в C++ не могут быть значениями-без-адреса. Компилятор размещает байты строки в сегменте статических данных (обычно .rodata), и выражение "abcd" обозначает именно этот массив в памяти. Следовательно, у него есть идентичность (адрес), и это **lvalue**.

2.3 Rvalue-ссылки (T&&)

Для захвата и работы с rvalue-категориями (временными объектами и xvalues) в C++11 был введен новый ссылочный тип: **Rvalue reference**, обозначаемый синтаксисом T&&.

- `T&` (`lvalue reference`): вяжется только к `lvalue`. (Исключение: `const T&` может вязаться к `rvalue`, продлевая ему жизнь, но объект будет неизменяемым).
- `T&&` (`rvalue reference`): вяжется только к `rvalue`. Позволяет изменять временный объект (например, "обворачивать" его).

```

1 int GetValue() { return 42; } // Возвращает prvalue
2
3 void demo() {
4     int i = 10;
5
6     int& ref1 = i;           // OK: i is lvalue
7     // int&& ref2 = i;       // ERROR: cannot bind rvalue ref to lvalue
8
9     // int& ref3 = GetValue(); // ERROR: cannot bind lvalue ref to prvalue
10    const int& ref4 = GetValue(); // OK: Lifetime Extension (но read-only)
11
12   int&& ref5 = GetValue(); // OK: ref5 ссылается на временный int(42)
13   ref5 = 100;             // Мы можем менять временный объект!
14 }
```

Важно!

Имя переменной – это всегда `lvalue`. Если у вас есть переменная типа `rvalue`-ссылка:

```

1 void foo(int&& x) {
2     // x имеет тип "rvalue reference to int"
3     // Но само выражение 'x' является lvalue!
4 }
```

У `x` есть имя, к нему можно обратиться, взять его адрес. Тот факт, что он ссылается на что-то, что было временным, не делает саму переменную `x` временной. Это критически важно для понимания того, почему внутри `move`-конструкторов нужно писать `std::move`.

2.4 Materialization (C++17)

До C++17 `prvalue` часто трактовалось как временный объект. C++17 изменил определение: `prvalue` – это "инициализатор", рецепт создания объекта. Временный объект (в памяти) создается только тогда, когда это необходимо. Этот процесс называется **Temporary Materialization**.

Пример:

```

1 struct Big { int data[100]; };
2
3 Big make() { return Big(); } // Возвращает prvalue (инициализатор)
4
5 void use() {
6     Big b = make();
7     // В C++17 здесь нет ни копирования, ни перемещения.
```

```
8     // prvalue из make() материализуется прямо в память переменной 'b'.
9     // Это гарантированная RVO (Return Value Optimization).
10 }
```

Материализация происходит, когда prvalue нужно превратить в glvalue (например, чтобы привязать к ссылке или обратиться к полю).

Резюме раздела

- **lvalue:** Есть имя, есть адрес. Нельзя мувить неявно.
- **prvalue:** Чистое значение, инициализатор. Нет адреса.
- **xvalue:** Эксп-lvalue, которое разрешили "грабить" (результат std::move).
- **glvalue:** lvalue + xvalue.
- **rvalue:** prvalue + xvalue.
- Переменная типа T&& сама по себе является **lvalue** в выражениях.

Глава 3

Семантика перемещения (Move Semantics): Механика и История

Семантика перемещения – это, пожалуй, самое значительное изменение в C++11, фундаментально изменившее подход к управлению ресурсами. Она позволяет не копировать данные, а передавать владение ими от одного объекта к другому. Чтобы понять, зачем это нужно, полезно взглянуть на историю языка.

3.1 Исторический контекст: Проблема лишних копий

До стандарта C++11 возврат тяжелых объектов из функций был сопряжен с накладными расходами. Рассмотрим классический пример фабричной функции, возвращающей вектор:

```
1 std::vector<int> LoadBigData() {
2     std::vector<int> data(1000000);
3     // ... заполнение данными ...
4     return data;
5 }
6
7 void Process() {
8     std::vector<int> my_data = LoadBigData();
9 }
```

В наивной реализации компилятора C++98 здесь происходило следующее:

1. Внутри LoadBigData создается локальный вектор (аллокация 4 МБ памяти).
2. При возврате создается временный объект-результат: вызывается конструктор копирования. Выделяются новые 4 МБ, данные копируются (`memcpy`). Локальный вектор уничтожается (освобождение 4 МБ).
3. В Process объект my_data инициализируется копированием из временного объекта. Еще одна аллокация, еще одно копирование. Временный объект уничтожается.

Итого: 3 аллокации, 2 глубоких копирования массивов. Хотя RVO (Return Value Optimization) существовала и раньше, она не была гарантирована во всех случаях (например, при сложной логике возврата или присваивании существующему объекту).

Трюк со swap

Чтобы избежать копирования, программисты использовали идиому swap. Вместо конструктора копирования "перемещение" имитировалось обменом внутреннего состояния с пустым объектом.

```

1 // Pre-C++11 стиль
2 std::vector<int> temp;
3 LoadBigData(temp); // Передача по ссылке (out-parameter)
4 std::vector<int> my_data;
5 my_data.swap(temp); // Обмен указателями за O(1)
```

Это работало быстро, но делало код громоздким и лишало функции возможности возвращать значения естественным образом.

3.2 Катастрофа std::auto_ptr

Попытка реализовать семантику владения до C++11 привела к созданию `std::auto_ptr`. Это был класс умного указателя, который пытался "перемещать" ресурс при копировании.

```

1 // Упрощенная логика auto_ptr
2 template <typename T>
3 struct auto_ptr {
4     T* ptr;
5
6     // "Копирующий" конструктор, который на самом деле перемещает
7     auto_ptr(auto_ptr& other) {
8         ptr = other.ptr;
9         other.ptr = nullptr; // МОДИФИКАЦИЯ ИСТОЧНИКА!
10    }
11 };
12
13 void bad_idea() {
14     std::auto_ptr<int> a(new int(10));
15     std::auto_ptr<int> b = a;
16     // Теперь 'b' владеет ресурсом, а 'a' стал nullptr.
17     // Это произошло при синтаксисе КОПИРОВАНИЯ.
18 }
```

Почему это плохо? Семантика копирования подразумевает создание независимого дубликата: оригинал не должен меняться. `auto_ptr` нарушал этот контракт. Это приводило к багам при использовании в стандартных контейнерах. Например, `std::sort` может брать "опорный" элемент копированием. В случае `auto_ptr` оригинал в массиве внезапно обнулялся, что приводило к потере данных и крашам. В C++11 `auto_ptr` был объявлен deprecated, а в C++17 удален из стандарта.

3.3 Философия Move Semantics

C++11 решил проблему, введя Rvalue-ссылки. Теперь мы можем перегружать функции для двух случаев:

1. const T& source — мы хотим **копировать** (источник важен, он останется неизменным).
2. T&& source — мы хотим **перемещать** (источник временный или нам явно разрешили его "разграбить").

Анатомия Move Constructor

Перемещающий конструктор "крадет" ресурсы у rvalue-объекта. Критически важно оставить источник в валидном состоянии, чтобы его деструктор отработал корректно.

Рассмотрим класс Holder, управляющий буфером памяти:

```

1 class Holder {
2     int* data_;
3     size_t size_;
4
5 public:
6     Holder(int size) : size_(size), data_(new int[size]) {}
7
8     ~Holder() {
9         delete[] data_; // Деструктор должен быть безопасным для nullptr
10    }
11
12    // Move Constructor
13    // Принимает неконстантную rvalue-ссылку
14    Holder(Holder&& other) noexcept
15        : data_(other.data_) // 1. Крадем указатель
16        , size_(other.size_) // 2. Крадем метаданные
17    {
18        // 3. Зануляем источник!
19        // Если этого не сделать, деструктор 'other' удалит память,
20        // которой теперь владеем мы. Будет double free.
21        other.data_ = nullptr;
22        other.size_ = 0;
23    }
24 };

```

Затраты: копирование двух скаляров (указатель + size). 0 аллокаций.

Анатомия Move Assignment Operator

Оператор присваивания сложнее, так как объект уже может владеть ресурсом, который нужно предварительно освободить. Также обязательна защита от самоприсваивания (`x = std::move(x)`).

```

1 Holder& operator=(Holder&& other) noexcept {
2     if (this == &other) {
3         return *this; // Защита от перемещения в себя
4     }
5

```

```

6   // 1. Освобождаем свой текущий ресурс
7   delete[] data_;
8
9   // 2. Крадем ресурсы
10  data_ = other.data_;
11  size_ = other.size_;
12
13  // 3. Зануляем источник
14  other.data_ = nullptr;
15  other.size_ = 0;
16
17  return *this;
18 }

```

3.4 std::move – это ложь

Одна из самых распространенных ошибок новичков – думать, что функция `std::move` что-то перемещает.

`std::move`

Это функция приведения типа. Она принимает объект любого типа (`lvalue` или `rvalue`) и безусловно приводит его к **rvalue-ссылке** (`T&&`).

Сам по себе вызов `std::move(x)` не генерирует никакого машинного кода для переноса данных. Он просто говорит компилятору: "Смотри на `x` как на временный объект". Перемещение происходит только тогда, когда результат `std::move` передается в конструктор или оператор присваивания, принимающий `T&&`.

```

1 std::vector<int> v1 = {1, 2, 3};
2 std::move(v1); // Ничего не происходит. v1 остался цел.
3
4 auto v2 = std::move(v1); // Вызывается vector(vector&&), вот теперь v1 пуст.

```

3.5 Ловушка реализации: Именованные Rvalue-ссылки

Это самый тонкий момент, на котором ошибаются даже опытные разработчики.

Важно!

Если у rvalue-ссылки есть имя, то она – lvalue.

Рассмотрим некорректную реализацию конструктора обертки:

```

1 class Wrapper {
2   std::string str_;
3 public:
4   // Мы принимаем строку как rvalue-ссылку (text)
5   explicit Wrapper(std::string&& text)

```

```

6       : str_(text) // ОШИБКА! Вызывается COPY constructor!
7   {}
8 };

```

Разбор ошибки:

1. Параметр `text` имеет тип `std::string&&`.
2. Внутри тела (или списка инициализации) функции, `text` – это переменная с именем. У нее есть адрес. Следовательно, выражение `text` имеет категорию значения **Ivalue**.
3. Конструктор `std::string` видит **Ivalue** и вызывает копирование (`const string&`).

Мы потеряли эффективность. Чтобы исправить это, нужно снова превратить **Ivalue** `text` в **Xvalue** с помощью `std::move`:

```

1 explicit Wrapper(std::string&& text)
2     : str_(std::move(text)) // Правильно: вызывается MOVE constructor
3 {}

```

3.6 Правило пяти (Rule of 5)

С появлением move-семантики "Правило трех" расширилось до "Правила пяти". Если классу требуется пользовательский деструктор (для освобождения ресурса), то, скорее всего, ему требуются:

1. Деструктор
2. Конструктор копирования
3. Оператор присваивания копированием
4. **Конструктор перемещения**
5. **Оператор присваивания перемещением**

Если вы не реализуете move-методы, компилятор не сгенерирует их автоматически, если у вас уже есть пользовательский деструктор или copy-методы. В этом случае класс будет только копируемым, что "убьет" производительность при работе с контейнерами.

Правило нуля (Rule of 0): Лучший способ написать класс – не управлять ресурсами вручную. Используйте `std::unique_ptr`, `std::vector`, `std::string`. У этих классов уже реализованы все 5 методов. Компилятор автоматически сгенерирует корректные дефолтные версии для вашего класса, которые просто вызовут соответствующие методы для полей.

```

1 class ModernHolder {
2     // std::vector сам управляет памятью.
3     // Нам не нужно писать ни деструктор, ни конструкторы.
4     // Все сгенерируется автоматически и будет поддерживать move.
5     std::vector<int> data_;
6 };

```

Резюме раздела

- Move-семантика заменяет глубокое копирование передачей указателей.
- `std::move` – это каст к `rvalue`, а не действие.
- Аргумент функции типа `T&&` внутри функции является `lvalue`. Используйте `std::move` для передачи его дальше.
- После перемещения объект находится в валидном, но неопределенном состоянии (`valid but unspecified`). Его можно уничтожить или присвоить ему новое значение, но читать из него нельзя.

Глава 4

Глава 4. Безопасность исключений и Контейнеры

Взаимодействие семантики перемещения с контейнерами стандартной библиотеки (в частности, `std::vector`) имеет нюанс, который часто упускают из виду. Этот нюанс касается гарантий безопасности исключений (Exception Safety Guarantees). Оказывается, наличие перемещающего конструктора не гарантирует, что вектор будет его использовать.

4.1 Проблема реаллокации вектора

Вспомним, как работает `std::vector::push_back` (или `emplace_back`), когда его емкость (`capacity`) исчерпана:

1. Выделяется новый блок памяти большего размера (обычно $\times 2$).
2. Элементы из старого блока переносятся в новый.
3. Старый блок удаляется.

В C++98 элементы всегда **копировались**. Если конструктор копирования одного из элементов бросал исключение, вектор просто уничтожал уже созданные копии в новом буфере и освобождал его. Старый буфер оставался нетронутым. Это обеспечивало **Строгую гарантию исключений (Strong Exception Guarantee)**: операция либо выполняется успешно, либо не меняет состояние программы (транзакционность).

С появлением C++11 возникло желание **перемещать** элементы вместо копирования. Это намного быстрее. Но что, если перемещающий конструктор бросит исключение на середине процесса?

Сценарий сбоя:

1. Вектор начал перемещать элементы. Элементы 1 и 2 успешно перемещены в новый буфер. В старом буфере они теперь "пустые" (`moved-from`).
2. При перемещении Элемента 3 возникает исключение.
3. Мы должны откатить операцию ("unwind"), чтобы обеспечить Strong Guarantee.
4. Мы уничтожаем копии в новом буфере.
5. **Проблема:** Мы не можем "вернуть" Элементы 1 и 2 обратно в старый буфер, потому что операция обратного перемещения тоже может бросить исключение!

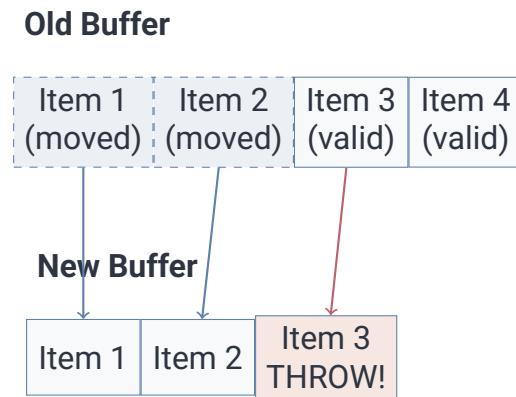


Рис. 4.1: Катастрофа при исключении во время Move

В итоге, у нас остались испорченные данные в старом буфере. Транзакционность нарушена.

4.2 Ключевое слово noexcept

Чтобы решить эту дилемму, `std::vector` использует стратегию: "Перемещать только если это безопасно, иначе копировать".

Как компилятор узнает, безопасно ли перемещать объект? Для этого используется спецификатор `noexcept` в объявлении перемещающего конструктора.

```

1 class SafeMover {
2 public:
3     // Мы гарантируем, что этот конструктор не бросит исключение
4     SafeMover(SafeMover&&) noexcept {
5         // ...
6     }
7
8     // Для копирования гарантий нет (может не хватить памяти)
9     SafeMover(const SafeMover&) {
10        // ...
11    }
12 };
13
14 class UnsafeMover {
15 public:
16     // noexcept отсутствует!
17     UnsafeMover(UnsafeMover&&) {
18         // ...
19     }
20     UnsafeMover(const UnsafeMover&) { /*...*/ }
21 };

```

При реаллокации `std::vector` проверяет свойство типа через `type traits` (в частности, `std::is_nothrow_move_constructible`)

Algorithm 1: Псевдокод логики реаллокации вектора

```

1 if std::is_nothrow_move_constructible<T>::value then
2     Move: Используем std::move для переноса элементов.:
3     Это быстро и безопасно (исключений не будет).:
4 else
5     if std::is_copy_constructible<T>::value then
6         Copy: Используем копирование.:
7         Медленно, но если будет исключение, старый буфер останется цел.:
8     else
9         Move (Unsafe): Если тип нельзя копировать (например, unique_ptr), у нас нет
          выбора. Рискуем и перемещаем.:

```

4.3 std::move_if_noexcept

Стандартная библиотека предоставляет утилиту std::move_if_noexcept, которая инкапсулирует эту логику выбора. Она возвращает rvalue-ссылку (разрешая мув) только если мув-конструктор помечен как noexcept, иначе она возвращает const lvalue-ссылку (форсируя копирование).

```

1 template <typename T>
2 void vector_realloc_stub(T* old_buf, T* new_buf, size_t size) {
3     for (size_t i = 0; i < size; ++i) {
4         // Магия выбора:
5         new (new_buf + i) T(std::move_if_noexcept(old_buf[i]));
6     }
7 }
```

4.4 Идиома "Move or Copy"

Из вышесказанного следует важная рекомендация для разработчиков классов:

Важно!

Всегда помечайте перемещающий конструктор и оператор присваивания как noexcept, если они не бросают исключений.

Обычно перемещение сводится к копированию указателей и скалярных типов, что никогда не бросает исключений. Если вы забудете noexcept, ваш класс будет работать корректно, но std::vector<YourClass> будет молча копировать элементы при расширении, что может катастрофически снизить производительность (с $O(1)$ до $O(N)$ глубоких копий).

Пример правильного объявления (Rule of 5 с noexcept):

```

1 class Efficient {
2     std::string data;
3 public:
4     Efficient(Efficient&& other) noexcept
5         : data(std::move(other.data)) {} // std::string move is noexcept

```

```
6     Efficient& operator=(Efficient&& other) noexcept {
7         data = std::move(other.data);
8         return *this;
9     }
10
11
12     // Copy operations (can throw std::bad_alloc)
13     Efficient(const Efficient&) = default;
14     Efficient& operator=(const Efficient&) = default;
15 };
```

Резюме раздела

- `std::vector` гарантирует Strong Exception Safety.
- При реаллокации вектор выбирает между Move и Copy.
- Выбор Move происходит **только** если Move Constructor помечен как `noexcept`.
- Если `noexcept` нет, происходит "молчаливое" копирование (Silent Pessimization).

Глава 5

Время жизни объектов (Object Lifetime) и Оптимизации

Управление временем жизни объектов – одна из самых сложных тем в C++. В отличие от языков с Garbage Collection, в C++ объект живет ровно столько, сколько определено его областью видимости (scope) или временем жизни контейнера. Однако существуют специальные правила, позволяющие продлевать жизнь временным объектам, а также оптимизации компилятора, которые могут эту жизнь "элиминировать" ради производительности.

5.1 Temporary Lifetime Extension

В C++ временные объекты (rvalue), созданные в выражении, обычно уничтожаются в конце выполнения этого "полного выражения" (full-expression), то есть на ближайшей точке с запятой.

Однако существует исключение: **Продление времени жизни (Lifetime Extension)**. Если временный объект привязывается к ссылке (константной lvalue-ссылке или rvalue-ссылке), то время жизни этого объекта продлевается до времени жизни самой ссылки.

```
1 struct Data {
2     ~Data() { std::cout << "Dead\n"; }
3 };
4
5 Data create() { return Data(); }
6
7 void demo() {
8     // 1. Обычное поведение
9     create();
10    std::cout << "After statement\n";
11    // Output: Dead -> After statement
12
13    // 2. Продление жизни
14    const Data& ref = create(); // Временный объект НЕ умирает здесь!
15    std::cout << "Using ref\n";
16    // Output: Using ref -> Dead (при выходе из scope)
17 }
```

Это позволяет безопасно работать с временными результатами, не копируя их.

5.2 Опасные паттерны и висячие ссылки (Dangling Refs)

Правило продления жизни имеет критическое ограничение: оно **не транзитивно**. Оно работает только при прямой привязке результата выражения к ссылке. Если временный объект "спрятан" внутри другого объекта или возвращается через цепочку функций, продление может не сработать или сработать не так, как вы ожидаете.

Ловушка 1: Возврат ссылки на временный объект

Это классическое UB, но с нюансом продления.

```

1 const int& get_ref(const int& x) { return x; }

2

3 void trap() {
4     // 1. Создается временный int(10).
5     // 2. Он передается в get_ref.
6     // 3. get_ref возвращает ссылку на него.
7     // 4. Ссылка привязывается к val.
8     // ПРОДЛЕНИЯ ЖИЗНИ НЕ ПРОИСХОДИТ!
9     // Правило работает только если ссылка инициализируется САМИМ prvalue,
10    // а здесь она инициализируется результатом функции (lvalue/xvalue).
11    const int& val = get_ref(10);
12
13    std::cout << val; // UB: обращение к памяти мертвого объекта
14 }
```

Ловушка 2: Доступ к полю временного объекта

Очень частая ошибка при использовании геттеров.

```

1 struct Wrapper {
2     std::string name;
3     const std::string& get_name() const { return name; }
4 };
5
6 Wrapper make_wrapper() { return Wrapper{"test"}; }
7
8 void disaster() {
9     // make_wrapper() создает временный Wrapper.
10    // Мы берем ссылку на его поле.
11    // В конце строки временный Wrapper уничтожается.
12    // Поле name уничтожается вместе с ним.
13    const std::string& s = make_wrapper().get_name();
14
15    std::cout << s; // UB! Ссылка 's' висит.
16 }
```

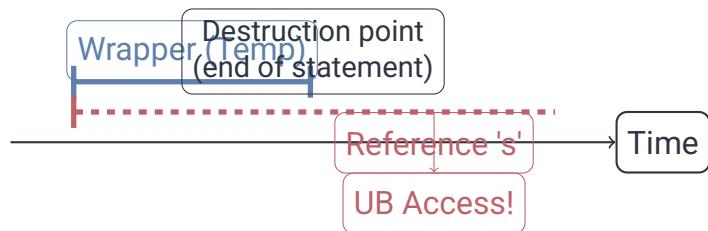


Рис. 5.1: Диаграмма времени жизни при доступе к полю временного объекта

5.3 RVO и Copy Elision

Компиляторы C++ обладают правом (а начиная с C++17 – обязанностью в некоторых случаях) полностью исключать создание временных объектов. Это называется **Return Value Optimization (RVO)**.

```

1 std::vector<int> generate() {
2     return std::vector<int>(1000); // prvalue
3 }
4
5 void use() {
6     std::vector<int> v = generate();
7 }
```

Без RVO: 1. Создается временный вектор внутри generate. 2. Он перемещается/копируется во временный результат возврата. 3. Результат перемещается/копируется в v.

С RVO (Copy Elision): Компилятор передает адрес переменной v внутрь функции generate. Вектор конструируется **прямо на месте** v. Никаких копий, никаких перемещений.

5.4 Pessimizing Move

С появлением std::move программисты начали писать его везде, думая, что помогают компилятору. В случае возврата значения это приводит к обратному эффекту – **Pessimizing Move**.

```

1 std::vector<int> bad_func() {
2     std::vector<int> local_vec;
3     // ...
4     // ОШИБКА! Мы явно приводим к rvalue-ссылке.
5     // Это ЗАПРЕЩАЕТ RVO, так как типы не совпадают (Obj vs Obj&&).
6     // Компилятор обязан вызвать Move Constructor.
7     return std::move(local_vec);
8 }
9
10 std::vector<int> good_func() {
11     std::vector<int> local_vec;
12     // ...
13     // ПРАВИЛЬНО! Работает NRVO (Named RVO).
14     // Объект будет построен сразу в целевой памяти.
15     return local_vec;
16 }
```

Важно!

Никогда не делайте `std::move` для локальной переменной в операторе `return`. Компилятор и так попробует применить NRVO. Если NRVO невозможно (сложная логика условий), компилятор автоматически применит `move` (как будто вы написали `std::move`). Ручной `std::move` здесь только мешает оптимизации.

Резюме раздела

- Ссылки могут продлевать жизнь временным объектам, но только "на один шаг".
- Остерегайтесь ссылок на части временных объектов.
- RVO/NRVO – самая мощная оптимизация в C++.
- `return std::move(x)` – это вредная привычка (Pessimizing Move).

Глава 6

Perfect Forwarding и Универсальные ссылки

Одной из самых мощных, но в то же время запутанных возможностей C++11 стала "Идеальная передача" (Perfect Forwarding). Эта механика позволяет писать обобщенные функции-обертки (wrappers), которые передают свои аргументы в другие функции **в точности** так, как они были получены: с сохранением категории значения (lvalue/rvalue) и cv-квалификаторов (const/volatile).

6.1 Проблема передачи аргументов

Представьте, что мы пишем фабричную функцию `make_shared`, которая должна создать объект типа `T`, передав аргументы в его конструктор.

Наивный подход с использованием константных ссылок:

```
1 template <typename T, typename Arg>
2 T* factory(const Arg& arg) {
3     return new T(arg);
4 }
```

Проблема: Если конструктор `T` принимает неконстантную ссылку (хочет изменить аргумент), этот код не скомпилируется. Если конструктор принимает rvalue-ссылку (move-семантика), мы передадим ей lvalue (так как `arg` имеет имя), и перемещения не произойдет.

Попытка перегрузки:

```
1 template <typename T, typename Arg>
2 void wrapper(Arg& arg) { func(arg); }
3
4 template <typename T, typename Arg>
5 void wrapper(const Arg& arg) { func(arg); }
```

Проблема: Количество перегрузок растет экспоненциально (2^N) от количества аргументов. Для 5 аргументов нужно написать 32 версии функции. Это неприемлемо.

6.2 Универсальные ссылки (Forwarding References)

Скотт Мейерс ввел термин "Universal Reference" (в стандарте C++17 закреплен термин **Forwarding Reference**), чтобы описать особый вид ссылок в шаблонах.

Forwarding Reference (T&&)

Если переменная или параметр объявлены как T&&, где T – это **выводимый** тип шаблона (deduced type), то это не rvalue-ссылка, а универсальная ссылка. Она может "превращаться" как в lvalue-ссылку, так и в rvalue-ссылку в зависимости от того, что передали на вход.

Синтаксис T&& работает по-разному в двух контекстах:

```

1 // 1. Rvalue Reference
2 void foo(int&& x); // Нет вывода типов. Это точно rvalue ref.
3
4 template <typename T>
5 void bar(std::vector<T>&& x); // T выводится, но && относится к vector, а не к T.
6
7 // 2. Forwarding Reference
8 template <typename T>
9 void func(T&& x); // T выводится. Это универсальная ссылка.
10
11 auto&& var = some_expression; // auto выводится. Это универсальная ссылка.

```

6.3 Правила схлопывания ссылок (Reference Collapsing)

Как компилятор понимает, во что превратить T&&? Это определяется механизмом дедукции типа и правилами схлопывания ссылок.

Когда мы передаем аргумент в шаблонную функцию func(T&& x):

- Если передан **lvalue** (например, int i), то T выводится как int&.
- Если передан **rvalue** (например, 42), то T выводится как int.

Теперь подставим выведенные типы в сигнатуру T&&. В C++ запрещены ссылки на ссылки (нельзя написать int& &), но компилятор может их генерировать в процессе инстанцирования. В этот момент вступают в силу правила "схлопывания":

Тип T	Сигнатура (T + &&)	Результат
int& (lvalue)	int& &&	int& (lvalue ref)
int (rvalue)	int &&	int&& (rvalue ref)

Таблица 6.1: Упрощенное правило: "Lvalue выигрывает всегда"

Полная таблица Collapsing Rules:

- T& + & → T&
- T& + && → T&
- T&& + & → T&
- T&& + && → T&&

Именно это позволяет одной сигнатуре `T&` принимать всё.

6.4 Механика `std::forward`

Теперь, когда у нас есть универсальная ссылка `x`, нам нужно передать её дальше. Но помните: **именованная rvalue-ссылка – это lvalue**. Если мы просто напишем `call(x)`, мы всегда будем передавать lvalue, теряя move-семантику. Нам нужен механизм, который делает `std::move`, но только если изначально пришло rvalue.

Эту роль выполняет `std::forward<T>`.

```
1 template <typename T>
2 void wrapper(T&& arg) {
3     // std::forward кастит arg обратно к его "изначальной" категории
4     target_func(std::forward<T>(arg));
5 }
```

Как это работает "под капотом"? Реализация `std::forward` выглядит примерно так (упрощенно):

```
1 // Если T = int& (передали lvalue)
2 // Возвращает: int& && -> int& (lvalue)
3 template <typename T>
4 T&& forward(typename remove_reference<T>::type& t) {
5     return static_cast<T&&>(t);
6 }
7
8 // Если T = int (передали rvalue)
9 // Возвращает: int&& (rvalue)
10 template <typename T>
11 T&& forward(typename remove_reference<T>::type& t) {
12     return static_cast<T&&>(t);
13 }
```

Таким образом, `std::forward` – это **условный cast**:

- Если `T` – ссылочный тип, он кастит к ссылке (lvalue).
- Если `T` – не ссылочный тип, он кастит к rvalue-ссылке (действует как `std::move`).

6.5 Практическое применение

1. Emplace-методы контейнеров

Методы `emplace_back` вектора используют perfect forwarding для конструирования элемента прямо в памяти контейнера, минуя создание временных объектов.

```
1 std::vector<std::string> v;
2
3 // Плохо: создание временного string, затем move
```

```
4 v.push_back(std::string("hello"));
5
6 // Идеально: аргумент "hello" пробрасывается в конструктор string,
7 // который вызывается прямо в памяти вектора (placement new).
8 v.emplace_back("hello");
```

2. make_unique / make_shared

Эти функции просто пересыпают все свои аргументы в конструктор целевого типа.

```
1 template<typename T, typename ... Args>
2 unique_ptr<T> make_unique(Args&&... args) {
3     return unique_ptr<T>(new T(std::forward<Args>(args)...));
4 }
```

Резюме раздела

- `T&&` в шаблоне – это Forwarding Reference, а не rvalue reference.
- Правила схлопывания ссылок гарантируют, что lvalue остается lvalue.
- `std::move` – безусловное приведение к rvalue.
- `std::forward<T>` – условное приведение, восстанавливающее исходную категорию значения.

Глава 7

Продвинутые идиомы C++ и Архитектурные паттерны

Завершая изучение семантики перемещения и управления памятью, мы рассмотрим продвинутые техники, которые позволяют выжать максимум из системы типов C++. Эти идиомы часто используются в библиотечном коде (например, STL или Boost) для оптимизации использования памяти и улучшения эргономики API.

7.1 Ref-qualifiers (Квалификаторы ссылок для методов)

Традиционно методы класса в C++ перегружались только по константности объекта (`this`):

```
1 struct Data {
2     std::string value;
3     const std::string& get() const { return value; } // Для const объектов
4     std::string& get() { return value; }           // Для non-const объектов
5 };
```

Однако, что если мы вызываем `get()` у временного объекта (`rvalue`)?

```
1 auto str = Data{"heavy_string"}.get();
```

В данном случае вызовется неконстантная версия `get()`, которая вернет lvalue-ссылку на поле. Затем произойдет **копирование** строки в переменную `str`, так как lvalue-ссылка не позволяет применить move-семантику. И это несмотря на то, что сам объект `Data` является временным и будет уничтожен сразу после этой строки! Мы упускаем возможность "украсть" данные.

C++11 позволяет перегружать методы по категории значения объекта `*this`, используя синтаксис `&` и `&&`.

```
1 struct ModernData {
2     std::vector<int> heavy;
3
4     // 1. Вызывается для lvalue (именованных объектов)
5     // Мы вынуждены вернуть const reference (копирование на стороне клиента)
```

```

6   const std::vector<int>& items() const& {
7     return heavy;
8   }
9
10 // 2. Вызывается для rvalue (временных объектов)
11 // Мы можем безопасно отдать ресурсы, так как объект скоро умрет
12 std::vector<int> items() && {
13   return std::move(heavy);
14 }
15 };
16
17 void demo() {
18   ModernData d;
19   auto v1 = d.items();           // Вызов (1): Копирование (безопасно)
20
21   auto v2 = ModernData{}.items(); // Вызов (2): Перемещение!
22   // v2 забирает буфер у временного вектора. 0 копий.
23 }
```

Этот механизм позволяет писать API, которые автоматически оптимизируются для временных объектов, предотвращая ненужные глубокие копии.

7.2 Empty Base Optimization (EBO)

В C++ любой объект должен иметь уникальный адрес. Из этого следует, что размер даже пустой структуры не может быть равен нулю.

```

1 struct Empty {};
2 static_assert(sizeof(Empty) >= 1); // Обычно 1 байт
```

Это создает проблему при композиции. Если мы включим пустой класс как поле в другой класс, он "съест" минимум 1 байт + возможный padding (выравнивание).

```

1 struct A {
2   Empty e; // 1 байт
3   int i;   // 4 байта
4   // Итоговый размер может быть 8 байт из-за выравнивания!
5 };
```

Однако стандарт разрешает оптимизацию: если пустой класс является **базовым**, он может иметь нулевой размер (его часть в layout объекта "схлопывается"). Это называется **Empty Base Optimization (EBO)**.

```

1 struct B : Empty { // EBO работает!
2   int i;
3 }
4 static_assert(sizeof(B) == sizeof(int)); // 4 байта
```

Где это используется? В аллокаторах, делетерах и компараторах STL. Например, `std::unique_ptr` хранит указатель и делетер. Если делетер – это пустая структура (без состояния, только

`operator()`, то благодаря EBO размер `unique_ptr` остается равным размеру сырого указателя. Если бы делегатор хранился как поле, размер указателя удвоился бы.

7.3 Опасность `const T&&`

Тип `const T&&` (константная rvalue-ссылка) является синтаксически валидным, но семантически бесполезным и даже вредным.

1. **Нельзя изменить:** Мы не можем "обворовать" объект, так как он `const`.
2. **Нельзя смувить:** Конструктор перемещения требует `T&&`, а не `const T&&`.

При попытке применить `std::move` к константному объекту происходит тихое "предательство":

```

1 const std::string s = "data";
2 // std::move(s) кастит к 'const string&&'
3 // Нет конструктора string(const string&&)
4 // Компилятор ищет ближайшее совпадение... string(const string&)
5 // ВЫЗЫВАЕТСЯ КОПИРОВАНИЕ!
6 std::string s2 = std::move(s);

```

Вывод: Никогда не используйте `const` для rvalue-ссылок и возвращаемых значений, если планируете перемещение. `const` блокирует move-семантику.

7.4 Destructive Move vs Non-destructive Move

C++ реализует **неразрушающее перемещение (Non-destructive Move)**. Это означает, что после перемещения исходный объект (*source*) остается жить. Он переходит в состояние "валидное, но неопределенное". Для него обязательно будет вызван деструктор.

- **Плюсы:** Совместимость с RAII. Деструктор всегда очищает ресурсы, даже у `moved-from` объекта (который просто ничего не делает, так как ресурсы обнулены).
- **Минусы:** Оверхед. Нам нужно вручную занулять указатели в источнике. Деструктор вызывается "вхолостую".

В языке **Rust**, напротив, реализовано **разрушающее перемещение (Destructive Move)**.

- Перемещение – это просто темсру битов.
- Исходный объект считается "мертвым" сразу после перемещения.
- Деструктор для исходного объекта **не вызывается**.
- Компилятор статически запрещает обращение к перемещенной переменной.

Подход Rust более эффективен (нет лишних записей нулей, нет лишних вызовов деструкторов), но требует более строгой системы типов (borrow checker), чтобы гарантировать безопасность на этапе компиляции. В C++ "зомби-объекты" (`moved-from`) – это плата за гибкость и совместимость с легаси-кодом.

Резюме раздела

- **Ref-qualifiers** позволяют перегружать методы для rvalue-объектов, оптимизируя цепочки вызовов.
- **EBO** позволяет "бесплатно" хранить stateless-объекты (политики, аллокаторы) внутри классов.
- `const` и `move` несовместимы.
- В C++ перемещение оставляет "пустую оболочку", которую нужно корректно уничтожить. В Rust перемещение уничтожает оболочку логически.