HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 01 – Введение. Память

Глава 1

Экосистема разработки и Инструментарий

Разработка на C++ существенно отличается от работы с интерпретируемыми языками (Python, JS) или языками с мощной стандартной экосистемой "из коробки" (Go, Rust). Здесь нет единого стандарта на сборку, управление пакетами или линтинг. Однако индустрия выработала набор инструментов де-факто, владение которыми является обязательным для написания качественного, переносимого и безопасного кода. В этой главе мы рассмотрим пайплайн сборки CMake, настройку IDE через LSP (ClangD) и инструменты динамического анализа памяти (Sanitizers).

Система сборки CMake

Нажатие кнопки "Play" в IDE скрывает за собой сложный процесс препроцессинга, компиляции каждого файла (Translation Unit), линковки объектных файлов и подключения библиотек. При переходе к проектам, состоящим из более чем одного файла, ручной вызов компилятора (например, `g++ main.cpp utils.cpp -o app`) становится неуправляемым.

> **CMake**
>
> Кроссплатформенная система мета-сборки. Она не собирает проект сама, а генерирует инструкции для нативной системы сборки (Makefiles, Ninja, Visual Studio Project и т.д.) на основе конфигурационного файла `CMakeLists.txt`.

Процесс работы с CMake всегда состоит из двух этапов:

1. **Конфигурация (Configuration):** CMake читает `CMakeLists.txt`, проверяет наличие компилятора, библиотек и генерирует дерево сборки.
2. **Сборка (Build):** Вызов нативного инструмента (например, `make`) для непосредственной компиляции исходного кода.

Пример минимального `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Добавляем цель для сборки исполняемого файла
```

```
add_executable(my_app main.cpp utils.cpp)

# Пример подключения санитайзеров (опционально)
target_compile_options(my_app PRIVATE -fsanitize=address -g)
target_link_options(my_app PRIVATE -fsanitize=address)
```

Для запуска сборки в терминале используется следующий паттерн out-of-source build (сборка в отдельной директории, чтобы не засорять исходники):

```
# 1. Создаем директорию build и запускаем конфигурацию
# -S . указывает, что исходники в текущей папке
# -B build указывает, что артефакты сборки кладем в папку build
cmake -S . -B build

# 2. Запускаем непосредственно сборку
# --build build — абстрактная команда, вызывающая make/ninja внутри папки build
cmake --build build
```

> **Важно!**
>
> Никогда не редактируйте сгенерированные файлы (Makefiles) внутри папки `build`. Все изменения конфигурации должны производиться только в `CMakeLists.txt`. Если вам нужно пересобрать проект с нуля, безопасно просто удалить папку `build` и запустить шаг конфигурации заново.

Интегрированная среда разработки (IDE) и LSP

Для комфортной работы с C++ (автодополнение, навигация по коду, подсветка ошибок) обычного текстового редактора недостаточно. Современный стандарт — использование протокола Language Server Protocol (LSP).

ClangD vs Стандартное расширение C++

В редакторе VS Code по умолчанию предлагается расширение от Microsoft (C/C++). Однако для Unix-систем и сложного C++ кода предпочтительнее использовать **clangd**.

- **Стабильность:** Clangd использует тот же фронтенд, что и компилятор Clang. Это гарантирует, что если код подсвечивается как ошибочный, он действительно не скомпилируется.
- **Скорость:** Индексация проекта происходит быстрее и точнее.

Для того чтобы Clangd (и любой другой языковой сервер) понимал структуру вашего проекта, ему нужен файл `compile_commands.json`. Этот файл содержит точные команды, которые используются для компиляции каждого файла (включая флаги, пути к заголовочным файлам и макросы).

Чтобы CMake сгенерировал этот файл, необходимо добавить флаг при конфигурации:

```
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

После этого файл `compile_commands.json` появится в папке `build`. Для корректной работы IDE часто требуется создать символическую ссылку на этот файл в корень проекта или настроить IDE на поиск файла в подпапках.

На заметку

Если IDE подчеркивает валидный код красным (например, не видит `<iostream>`), в 99% случаев проблема в отсутствии или некорректности `compile_commands.json`.

Динамический анализ: AddressSanitizer (ASan)

Ошибки работы с памятью — самый коварный класс багов в C++. Они могут приводить к неопределенному поведению (UB), которое не проявляется при локальном запуске, но роняет продакшн. Статический анализ не может поймать все ошибки. Здесь на помощь приходят санитайзеры.

> **AddressSanitizer (ASan)**
>
> Инструмент динамического анализа, который встраивает проверки доступа к памяти непосредственно в исполняемый файл на этапе компиляции.

Механизм работы

При компиляции с флагом `-fsanitize=address` компилятор инструментирует код, окружая каждый выделенный объект в памяти (как на стеке, так и в куче) специальными "красными зонами" (redzones). При любом обращении к памяти программа проверяет, не попадает ли адрес в красную зону. Если попадает — выполнение немедленно прерывается с подробным отчетом об ошибке.

ASan позволяет обнаруживать:

- Выход за границы массива (Out-of-bounds access) на стеке и в куче.
- Использование памяти после освобождения (Use-after-free).
- Двойное освобождение (Double-free).
- Утечки памяти (Memory leaks) — работает как LeakSanitizer.

Стоимость использования

Использование ASan замедляет выполнение программы примерно в 2 раза и увеличивает потребление памяти. Это приемлемо для этапа тестирования и отладки (Debug builds), но обычно отключается в финальной сборке (Release builds), где важна максимальная производительность.

Пример включения в `CMakeLists.txt` (рекомендуется выносить в отдельный тип сборки):

```
if(ENABLE_ASAN)
    add_compile_options(-fsanitize=address -g)
    add_link_options(-fsanitize=address)
endif()
```

Отладка: GDB

Хотя современные IDE предоставляют графические интерфейсы для отладки, умение работать с консольным отладчиком GDB (GNU Debugger) является критическим навыком.

- **Работа на сервере:** Если ваше приложение упало на удаленном сервере (headless environment), у вас не будет доступа к GUI. GDB — единственный способ проанализировать `core dump` (слепок памяти упавшего процесса).
- **Автоматизация:** GDB поддерживает скриптинг. Вы можете написать сценарий, который автоматически выполняет 100 итераций цикла и останавливается только при выполнении сложного условия, что утомительно делать "прокликиванием" в GUI.

Базовый сеанс отладки в GDB:

```
$ gdb ./my_app
(gdb) break main          # Поставить точку останова на main
(gdb) run                 # Запустить программу
(gdb) next                # Следующая строка (шаг без захода внутрь)
(gdb) step                # Шаг с заходом внутрь функции
(gdb) print variable_name # Вывести значение переменной
(gdb) continue            # Продолжить выполнение до следующей точки
(gdb) bt                  # Backtrace: показать стек вызовов
```

Резюме раздела

Грамотно настроенное окружение — половина успеха.

1. Используйте **CMake** для управления сборкой, разделяя исходный код и артефакты.
2. Настройте **ClangD** и генерацию `compile_commands.json` для умного автодополнения.
3. Всегда собирайте отладочные версии с **AddressSanitizer** (`-fsanitize=address`), чтобы ловить ошибки памяти на ранних этапах.
4. Изучите основы **GDB** для отладки в сложных окружениях.

Глава 2

Модель памяти и Жизненный цикл объектов

Понимание работы с памятью — фундаментальный навык разработчика на C++. В отличие от языков с управляемой памятью (Java, Python, Go), где о расположении объектов заботится виртуальная машина и сборщик мусора, C++ предоставляет прямой доступ к управлению ресурсами. Это дает огромную власть, но налагает не меньшую ответственность: по статистике, около 70-80% всех уязвимостей в безопасности ПО связаны именно с ошибками работы с памятью (memory safety issues).

В этой главе мы рассмотрим, что такое память с точки зрения стандарта C++, что такое "объект" и какие существуют классы хранения данных.

Фундаментальная модель памяти

Миф о "Большом массиве байт"

Интуитивно память часто представляют как единый, непрерывный массив байт, пронумерованный от 0 до N. На физическом уровне (RAM) это отчасти верно, однако программа на C++ работает не с физическими чипами, и даже не напрямую с виртуальной памятью операционной системы. Она работает с **Абстрактной машиной C++**.

> **Важно!**
>
> С точки зрения стандарта C++, память — это не "просто массив", а набор доступных **объектов**. Доступ к памяти, где не создан объект (или его время жизни истекло), является неопределенным поведением (Undefined Behavior).

Операционная система предоставляет процессу *виртуальное адресное пространство*. Это пространство изолировано: программа не может случайно (или специально) прочитать данные соседнего процесса (например, браузера), просто обратившись по адресу $0x1234$. Адресное пространство процесса фрагментировано: оно состоит из сегментов кода, данных, стека и кучи, между которыми могут находиться огромные "дыры" невыделенной памяти.

Байт и слово

Минимальной адресуемой единицей памяти в C++ является байт.

- `sizeof(char)` всегда равен 1. Это определение единицы измерения в языке.
- Количество бит в байте определяется макросом `CHAR_BIT` из заголовка `<climits>`.

На заметку

Формально стандарт требует, чтобы `CHAR_BIT` $\geqslant$ `8`. Исторически существовали архитектуры с 9-битными или 16-битными байтами, но в современной индустрии (x86, ARM) байт всегда равен 8 битам (октету). Тем не менее, полагаться на хардкод константы 8 — плохой тон; лучше использовать `CHAR_BIT`.

Анатомия Объекта

В C++ всё есть объект (или ссылка на него). Понимание свойств объекта критично для избежания UB.

> **Объект в C++**
>
> Объект — это область памяти (region of storage), которая имеет:
>
> - **Size (Размер):** определяется `sizeof(T)`.
> - **Alignment (Выравнивание):** требование процессора к адресу начала объекта (например, `int` часто должен лежать по адресу, кратному 4).
> - **Storage Duration (Тип хранения):** где и сколько живет объект.
> - **Lifetime (Время жизни):** интервал времени выполнения, когда объект существует.
> - **Type (Тип):** интерпретация битов в этой памяти.
> - **Value (Значение):** конкретное содержимое.

Ключевой концепт: **Lifetime**.

1. Память выделяется (allocation).
2. Отрабатывает конструктор — начинается lifetime.
3. ... Использование объекта ...
4. Отрабатывает деструктор — заканчивается lifetime.
5. Память освобождается (deallocation).

Любое использование указателя или ссылки на объект до шага 2 или после шага 4 — это ошибка **Use-after-free** или использование неинициализированной памяти.

Категории хранения (Storage Durations)

Стандарт выделяет четыре типа продолжительности хранения. Понимание различий между ними позволяет отвечать на вопрос: "Когда умрет эта переменная?".

1. **Automatic (Автоматическая):** Локальные переменные функций, аргументы. Живут на стеке.

2. **Static (Статическая):** Глобальные переменные, `static` поля классов, `static` переменные внутри функций. Живут всё время работы программы.
3. **Dynamic (Динамическая):** Объекты, созданные через `new`/`malloc`. Живут пока их явно не удалят. (Подробнее в следующей главе).
4. **Thread-local (Локальная для потока):** Живут пока жив поток, создавший их (спецификатор `thread_local`).

Автоматическая память (Stack)

Самый быстрый и безопасный тип памяти. Обычно называется "стеком", так как реализуется через аппаратный стек процессора (регистры SP/ESP/RSP).

Принцип работы

При входе в функцию (scope) указатель стека сдвигается, выделяя место под все локальные переменные разом. При выходе — сдвигается обратно.

- **Аллокация:** мгновенная (просто сложение регистра с константой).
- **Деаллокация:** автоматическая при выходе из скоупа (закрывающая фигурная скобка `}`).

> **Важно!**
>
> Порядок разрушения объектов на стеке **строго обратен** порядку их создания. Это принцип LIFO (Last In, First Out).

```cpp
void logic() {
    std::string a = "First";
    std::vector<int> b = {1, 2};
    // ... работа ...
} // Сначала вызывается ~vector (b), затем ~string (a)
```

Это свойство является основой идиомы RAII (Resource Acquisition Is Initialization), о которой мы поговорим позже.

Проблема Stack Overflow (Задача с семинара)

Размер стека ограничен. В Linux это обычно 8 MiB, в Windows — 1-2 MiB (настраивается при компиляции). Исчерпание стека приводит к аварийному завершению программы (Segmentation Fault).

Рассмотрим задачу с семинара, демонстрирующую опасность рекурсии:

```cpp
void recursive_boom(int depth) {
    // Массив выделяется на стеке в КАЖДОМ кадре рекурсии
    char buffer[1024];

    // Предотвращение оптимизации (чтобы компилятор не выкинул массив)
    buffer[0] = depth % 256;
```

```
7
8       if (depth > 0) {
9           recursive_boom(depth - 1);
10      }
11  }
12
13  int main() {
14      recursive_boom(100000); // Глубина 100 тысяч
15  }
```

Анализ: Каждый вызов функции `recursive_boom` создает новый стековый кадр (stack frame). В этом кадре хранится:

1. Адрес возврата (8 байт на x64).
2. Аргумент `depth` (4-8 байт с учетом выравнивания).
3. Массив `buffer` (1024 байта).

Итого один вызов занимает чуть больше 1 Кб. При глубине 100 000 рекурсивных вызовов мы пытаемся занять $100\,000 \times 1$ KiB ≈ 100 MiB. Это значительно превышает лимит стека (8 MiB), что гарантированно приводит к краху программы.

На заметку

Крупные объекты (массивы на миллион элементов, тяжелые структуры) никогда не должны создаваться на стеке. Используйте `std::vector` или умные указатели для размещения их данных в куче.

Статическая память (Static)

Переменные со статической продолжительностью хранения инициализируются один раз (обычно до вызова `main` или при первом проходе управления) и разрушаются при завершении программы.

Сюда относятся:

- Глобальные переменные.
- Переменные, объявленные как `static` внутри функций или классов.

Static Initialization Order Fiasco

Самая большая проблема глобальных переменных — неопределенный порядок инициализации между разными единицами трансляции (cpp-файлами).

Представьте два файла:

```
1  // File A.cpp
2  int x = func(); // func() возвращает 42
3
4  // File B.cpp
5  extern int x;
```

```
int y = x + 1;
```

Стандарт C++ **не гарантирует**, что `A.cpp` будет инициализирован раньше `B.cpp`. Если инициализация `B` произойдет первой, переменная `x` будет иметь значение 0 (zero-initialization), и `y` станет равно 1. Если порядок будет обратным, `y` станет 43. Это классический "плавающий" баг, зависящий от фазы луны и линковщика.

Решение: Meyers Singleton

Для безопасной работы с глобальными состояниями (если они необходимы) Скотт Мейерс предложил паттерн, использующий `static` переменную *внутри функции*.

Стандарт C++11 гарантирует, что статические переменные внутри функции инициализируются:

1. Ровно один раз.
2. В момент, когда поток управления **впервые** проходит через их объявление.
3. Потокобезопасно (компилятор вставляет неявные блокировки/guard variables).

```
class Database {
    // ... подключение к БД ...
};

Database& GetDB() {
    // Инициализация произойдет только при первом вызове GetDB()
    static Database db_instance;
    return db_instance;
}

void ClientCode() {
    // Гарантированно инициализированный объект
    GetDB().query("SELECT * FROM users");
}
```

Этот подход решает проблему порядка инициализации (ленивая инициализация по требованию) и является стандартным способом реализации синглтонов в современном C++.

Резюме раздела

- Память в C++ — это набор объектов с временем жизни, а не просто байты.
- **Стек (Automatic)**: быстро, автоматически чистится, но размер ограничен. LIFO.
- **Глобальные (Static)**: живут вечно, опасны из-за порядка инициализации. Используйте локальные статики (Meyers Singleton).
- Избегайте глубокой рекурсии и больших массивов на стеке во избежание **Stack Overflow**.

Глава 3

Динамическая память и RAII

Мы разобрали автоматическую (стековую) и статическую память. Однако стека недостаточно для решения большинства реальных задач: его размер мал (мегабайты), а время жизни объектов жестко привязано к области видимости. Для работы с большими объемами данных (изображения, базы данных, сетевые буферы) или объектами, время жизни которых определяется бизнес-логикой, используется динамическая память (Heap или "Куча").

В "старом" C++ (до C++11) и в языке C работа с кучей требовала ручного микроменеджмента. В современном C++ мы используем идиому RAII, чтобы автоматизировать этот процесс.

Низкоуровневые механизмы: malloc vs new

В C++ существуют два способа выделить сырую память. Важно понимать разницу между ними, так как смешивать их — грубая ошибка.

1. **C-style (`malloc/free`):** Функции из стандартной библиотеки C (`<cstdlib>`).
    - **Типизация:** Возвращают `void*`. Требуют приведения типов.
    - **Объекты:** Только выделяют байты. **Не вызывают конструкторы** и деструкторы.
    - **Ошибки:** При нехватке памяти возвращают `nullptr`.
2. **C++ Operators (`new/delete`):** Операторы языка.
    - **Типизация:** Типизированы, возвращают `T*`.
    - **Объекты:** Сначала выделяют память (через `operator new`), затем **вызывают конструктор**. Оператор `delete` вызывает деструктор, затем освобождает память.
    - **Ошибки:** При нехватке памяти бросают исключение `std::bad_alloc`.

> **Важно!**
>
> В C++ использование `malloc` практически всегда является ошибкой, кроме случаев взаимодействия с legacy C-библиотеками или написания собственных аллокаторов. Для создания объектов всегда используется `new` (или его обертки).

Проблема Exception Safety

Почему ручные `new` и `delete` считаются плохим тоном ("моветоном")? Главная причина — сложность написания кода, устойчивого к исключениям (Exception Safety).

Рассмотрим классический пример утечки ресурсов, который сложно заметить невооруженным глазом:

```
void process(Widget* w1, Widget* w2);

void problematic_usage() {
    // ОПАСНО: Порядок вычисления аргументов не определен (до C++17)
    // Даже в C++17, если конструктор второго Widget кинет исключение,
    // первый Widget уже создан, но указатель на него потерян.
    process(new Widget(1), new Widget(2));
}
```

Разбор сценария утечки: Компилятор может сгенерировать код в таком порядке:

1. Выделение памяти под первый Widget.
2. Вызов конструктора первого Widget (успех). Указатель лежит во временном регистре процессора.
3. Выделение памяти под второй Widget.
4. **Вызов конструктора второго Widget — бросает исключение!**

В этот момент стек начинает разматываться (stack unwinding). Мы вылетаем из функции `problematic_usage`. Но указатель на первый созданный Widget нигде не был сохранен в переменную, у которой есть деструктор. Адрес потерян. Память утекла.

Чтобы исправить это на "сырых" указателях, пришлось бы писать громоздкий код:

```
void safe_but_ugly() {
    Widget* w1 = new Widget(1);
    try {
        Widget* w2 = new Widget(2);
        try {
            process(w1, w2);
        } catch (...) {
            delete w2;
            throw;
        }
    } catch (...) {
        delete w1; // Очистка w1 в случае ошибки создания w2
        throw;
    }
    delete w2;
    delete w1;
}
```

Очевидно, что писать так — невозможно. Здесь на помощь приходит RAII.

Идиома RAII (Resource Acquisition Is Initialization)

Это, пожалуй, самая важная идиома C++. Идея гениальна в своей простоте: **связать время жизни ресурса (память, файл, мьютекс) с временем жизни объекта на стеке**.

> **RAII**
>
> Захват ресурса происходит в конструкторе объекта, а освобождение — в его деструкторе. Поскольку объекты на стеке гарантированно разрушаются при выходе из области видимости (в том числе при вылете исключения), утечки ресурсов становятся невозможными.

Вместо сырого указателя мы используем "умный" объект-обертку. Стандартная библиотека предоставляет готовое решение.

Умный указатель std::unique_ptr

`std::unique_ptr<T>` (из заголовка `<memory>`) — это легковесная обертка над сырым указателем.

- **Семантика:** Я владею объектом единолично.
- **Конструктор:** Принимает сырой указатель.
- **Деструктор:** Вызывает `delete` для хранимого указателя.
- **Копирование:** Запрещено (удален copy constructor).
- **Перемещение:** Разрешено (move constructor передает владение).

Исправим пример с `process` используя современный подход:

```cpp
#include <memory>

void process(std::unique_ptr<Widget> w1, std::unique_ptr<Widget> w2);

void safe_usage() {
    // std::make_unique создает объект и сразу возвращает unique_ptr.
    // Если make_unique для w2 кинет исключение,
    // unique_ptr для w1 уже будет создан и корректно уничтожится.
    process(
        std::make_unique<Widget>(1),
        std::make_unique<Widget>(2)
    );
}
```

> **На заметку**
>
> Функция `std::make_unique<T>(args...)` появилась в C++14. Всегда предпочитайте её использованию `new`. Она не только безопаснее (исключает утечку, описанную выше), но и избавляет от необходимости писать имя типа `T` дважды.

Массивы: new[] и delete[]

В C++ одиночные объекты и массивы объектов — это разные сущности с точки зрения управления памятью.

- `new T` требует `delete ptr`.
- `new T[N]` требует `delete[] ptr`.

> **Важно!**
>
> Перепутать операторы удаления — это Неопределенное поведение (UB). Если вы выделили массив через `new int[10]`, а удалили через `delete`, программа может упасть или повредить кучу (Heap Corruption).

Почему? Обычно при выделении массива `new[]` компилятор сохраняет размер массива (например, 4 байта перед началом возвращаемого указателя), чтобы знать, сколько раз вызвать деструктор. Оператор `delete[]` читает этот размер и идет в цикле. Обычный `delete` просто удаляет один объект.

В современном C++ сырые массивы в динамической памяти (`new[]`) практически не нужны. Используйте:

- `std::vector<T>` — для динамических массивов с изменяемым размером.
- `std::string` — для строк.

Резюме раздела

1. Никогда не используйте `malloc` в C++ коде.
2. Избегайте явных вызовов `new` и `delete`.
3. Используйте `std::unique_ptr` и `std::make_unique` для управления одиночными объектами.
4. Используйте `std::vector` вместо динамических массивов `new T[]`.
5. **RAII** — ваш главный инструмент борьбы с утечками. Ресурс должен принадлежать объекту на стеке.

Глава 4

Низкоуровневая работа с памятью: Placement New и Выравнивание

До этого момента мы рассматривали стандартные способы создания объектов: на стеке (автоматически) или в куче (через `new` или `std::make_unique`). В обоих случаях компилятор и рантайм берут на себя две задачи одновременно:

1. Выделение сырой памяти (allocation).
2. Конструирование объекта в этой памяти (construction).

Однако в системном программировании, при написании собственных контейнеров (например, `std::vector`) или аллокаторов, эти этапы необходимо разделять. Нам нужно уметь создать объект в *уже выделенной* памяти по конкретному адресу. Этот механизм называется **Placement New**.

Это "черная магия" C++, открывающая ящик Пандоры с неопределенным поведением. Ошибиться здесь легко, а цена ошибки — трудноуловимые баги, связанные с выравниванием (alignment) и временем жизни.

Разделяющий властвуй: Placement New

Обычный оператор `new T(...)` делает две вещи: вызывает функцию выделения памяти (`operator new`), а затем вызывает конструктор. Placement New — это перегрузка оператора `new`, которая принимает дополнительный аргумент (адрес памяти) и **не выделяет память**, а просто вызывает конструктор по этому адресу.

Синтаксис и Жизненный цикл

Синтаксис выглядит так:

```
new (адрес) Тип(аргументы);
```

Для использования необходимо подключить заголовок `<new>`.

Рассмотрим полный цикл жизни объекта, созданного вручную. Поскольку память не выделялась через обычный `new`, мы не имеем права вызывать `delete`. Мы обязаны вызвать деструктор вручную.

```cpp
#include <new>
#include <string>
#include <iostream>

int main() {
    // 1. Выделяем сырую память (достаточную для хранения string)
    // Внимание: здесь есть проблема с выравниванием (см. далее)!
    char buffer[sizeof(std::string)];

    // 2. Конструируем объект в буфере (Placement New)
    std::string* s_ptr = new (buffer) std::string("Hello, Placement New!");

    // 3. Используем объект
    std::cout << *s_ptr << ", length: " << s_ptr->size() << std::endl;

    // 4. ЯВНО вызываем деструктор
    s_ptr->~basic_string();

    // 5. Память буфера освободится автоматически (так как это стек)
    // Если бы buffer был выделен malloc-ом, нужно было бы сделать free(buffer).
}
```

Важно!

Никогда не вызывайте `delete s_ptr` для объекта, созданного через placement new на стековом буфере или внутри другого объекта. Оператор `delete` попытается передать адрес буфера системному аллокатору памяти, который понятия не имеет об этом адресе (если это стек) или уже считает его занятым. Это гарантированный крэш или коррупция кучи.

Выравнивание (Alignment)

В примере выше (`char buffer[...]`) мы допустили критическую ошибку, которая на архитектуре x86 может остаться незамеченной, но приведет к падению программы (SIGBUS) на ARM или SPARC, а также к существенному падению производительности.

Почему sizeof недостаточно?

Процессор работает с памятью не побайтово, а машинными словами (4, 8, 16 байт). Для эффективного (а иногда и корректного) доступа к данным адреса должны быть кратны размеру типа.

- `int` (4 байта) обычно должен лежать по адресу, кратному 4.
- Указатели (8 байт на 64-bit) — по адресу, кратному 8.
- Векторные инструкции (SIMD, SSE/AVX) могут требовать выравнивания по 16, 32 или 64 байтам.

Массив `char buffer[N]` имеет выравнивание 1 (так как `alignof(char) == 1`). Он может начаться по любому адресу (например, 0x1001). Тип `std::string` внутри себя содержит указатели и счетчики размеров, требующие выравнивания 8 (на 64-битных системах).

Если мы разместим `std::string` по адресу 0x1001, мы нарушим требования выравнивания (**Misaligned Access**). Это **Undefined Behavior**.

Решение: alignas и alignof

Чтобы исправить это, буфер должен быть выровнен так же, как и тип, который мы собираемся в него положить.

- `alignof(T)` — возвращает требование к выравниванию для типа `T`.
- `alignas(N)` — спецификатор, заставляющий компилятор выровнять переменную по границе `N`.

Правильная реализация буфера:

```
template <typename T>
struct AlignedStorage {
    // Выравниваем массив байт так же, как тип T
    alignas(T) std::byte data[sizeof(T)];

    // Вспомогательные методы для удобства
    T* get() { return reinterpret_cast<T*>(data); }
    const T* get() const { return reinterpret_cast<const T*>(data); }
};

int main() {
    AlignedStorage<std::string> storage;

    // Теперь адрес storage.data гарантированно кратен alignof(string)
    std::string* s = new (storage.get()) std::string("Safe!");

    s->~basic_string();
}
```

Также в стандарте C++ (заголовок `<type_traits>`) есть `std::aligned_storage`, но начиная с C++23 он объявлен устаревшим (deprecated) в пользу использования `alignas` вручную, как показано выше, из-за сложности правильного использования API.

Тривиальная разрушаемость (Trivially Destructible)

Обязательно ли всегда вызывать деструктор при ручном управлении памятью?

```
{
    char buf[sizeof(int)];
    int* p = new (buf) int(42);
    // Нужно ли звать p->~int()?
}
```

Для фундаментальных типов (`int`, `float`, указатели) и простых C-структур (POD) деструктор ничего не делает (no-op). Вызов `p->~int()` корректен синтаксически, но компилятор его просто вырежет.

Существует трейт (свойство типа) `std::is_trivially_destructible_v<T>`.

- Если **true**: Вызов деструктора можно безопасно пропустить. Память можно просто "забыть" или перезаписать.
- Если **false** (например, `std::string`, `std::vector`): Вызов деструктора обязателен. Если его пропустить, утекут ресурсы, которыми владел объект (внутренние буферы в куче).

> **Важно!**
>
> Оптимизация с пропуском деструкторов широко используется в реализации `std::vector::clea` Если элементы тривиально разрушаемы, вектор просто обнуляет свой размер (`size = 0`), не пробегая циклом по всем элементам. Это дает колоссальный прирост производительности.

Практический пример: простейший Any

Чтобы закрепить материал, напишем упрощенный класс, который может хранить либо `int`, либо `double` (похоже на `union`, но с активным переключением).

```
#include <iostream>
#include <new>

struct NumberHolder {
    // Достаточно места для самого большого типа
    // Выравнивание по максимуму из двух
    alignas(double) std::byte storage[sizeof(double)];

    enum Type { INT, DOUBLE, NONE } currentType = NONE;

    void setInt(int val) {
        destroyCurrent(); // Сначала очищаем старое
        new (storage) int(val);
        currentType = INT;
    }

    void setDouble(double val) {
        destroyCurrent();
        new (storage) double(val);
        currentType = DOUBLE;
    }

    void destroyCurrent() {
        if (currentType == INT) {
            reinterpret_cast<int*>(storage)->~int(); // Тривиально, но для
            ↪ порядка
        } else if (currentType == DOUBLE) {
            reinterpret_cast<double*>(storage)->~double();
        }
        currentType = NONE;
    }

    ~NumberHolder() {
        destroyCurrent();
```

```
34         }
35 };
```

Резюме раздела

1. **Placement New** позволяет конструировать объекты в заранее выделенной памяти. Он не выделяет память сам.
2. Объекты, созданные через placement new, требуют явного вызова деструктора: `ptr→T()`. Использование `delete` запрещено.
3. **Выравнивание** критически важно. Простого массива `char` недостаточно. Используйте `alignas(T)`.
4. Доступ к невыровненным данным — это Undefined Behavior.
5. Для типов с `is_trivially_destructible` вызов деструктора можно опустить (оптимизация).

Глава 5

Оптимизация строк: SSO и String View

Строки — один из самых часто используемых типов данных в прикладном программировании. Поэтому эффективность реализации `std::string` критически важна для общей производительности C++ приложений. В этой главе мы разберем две важнейшие концепции: оптимизацию коротких строк (SSO), которая "прячет" аллокации, и `std::string_view` — инструмент для эффективной работы с подстроками без копирования.

Анатомия std::string и SSO

Наивная реализация строки могла бы выглядеть как простая обертка над динамическим массивом:

```
class NaiveString {
    char* data_;
    size_t size_;
    size_t capacity_;
};
```

На 64-битной архитектуре размер такого объекта составляет $8+8+8 = 24$ байта. Сама строка хранится в куче (heap). Это означает, что даже для строки "hello" (5 байт) мы вынуждены делать дорогостоящий системный вызов `malloc/new`, ловить cache miss при обращении к данным и фрагментировать память.

Учитывая, что в реальных программах огромное количество строк очень короткие (имена, ключи в JSON, идентификаторы), разработчики стандартной библиотеки применили оптимизацию **SSO (Small String Optimization)**.

Механика SSO

Идея SSO заключается в использовании памяти самого объекта (этих 24 байт на стеке) для хранения содержимого строки, если оно туда помещается.

Внутреннее устройство `std::string` можно представить как объединение (`union`) двух состояний:

1. **Long String (Куча):** Классическая схема (Pointer, Size, Capacity).
2. **Short String (SSO):** Буфер символов прямо внутри структуры + 1 байт для размера.

Псевдокод реализации (упрощенно):

```
class String {
    static const size_t SSO_CAPACITY = 23; // 24 байта - 1 байт на размер/null

    union {
        // Режим "Длинная строка"
        struct {
            char* ptr;
            size_t size;
            size_t capacity;
        } heap_buf;

        // Режим "Короткая строка"
        struct {
            char buffer[SSO_CAPACITY];
            unsigned char size_byte; // Хранит размер и флаг режима
        } stack_buf;
    };

public:
    // ... методы ...
};
```

Как это работает?

Поскольку объект занимает 24 байта, мы можем хранить строку длиной до 23 символов **без аллокации памяти**.

- **23 символа:** полезная нагрузка.
- **24-й байт:** используется двояко. В режиме SSO он хранит размер оставшегося места (или длину). В режиме длинной строки это часть поля `capacity` или специальный флаг.

Компиляторы используют битовые хаки. Например, младший бит последнего байта может служить флагом:

- 0: Режим SSO.
- 1: Режим длинной строки (указатель).

> **На заметку**
>
> Благодаря SSO, создание `std::string s = "test";` не приводит к обращению к куче. Это делает работу с короткими строками такой же быстрой, как использование автоматических массивов `char[N]`.

std::string_view (C++17)

До C++17 существовала проблема передачи строк в функции.

- **Передача по значению (`void f(std::string s)`):** Всегда вызывает копирование (глубокую аллокацию), если строка длинная. Дорого.

- **Передача по константной ссылке (`void f(const std::string& s)`):** Избавляет от копирования. Но что, если мы хотим передать подстроку?

```
void process(const std::string& s);

std::string data = "Header: Value";
// Чтобы передать "Value", нужно создать временную строку (аллокация!)
process(data.substr(8));
```

Решением стал `std::string_view` — легковесная "вьюшка" (view), которая смотрит на кусок памяти, но не владеет им.

Структура и Производительность

`std::string_view` состоит всего из двух полей:

```
class string_view {
    const char* ptr_; // Указатель на начало
    size_t size_;     // Длина
};
```

Размер объекта — 16 байт (на 64-bit).

> **Важно!**
>
> **Передача параметров:** `std::string_view` принято передавать **по значению**, а не по ссылке.
>
> - `void f(std::string_view sv)`
>
> Почему? Объект размером 16 байт идеально ложится в два 64-битных регистра процессора (например, RDI и RSI в System V ABI). Передача через регистры быстрее, чем передача ссылки (которая суть указатель), так как избегается лишнее разыменование (double indirection) при доступе к полям `ptr_` и `size_`.

Операция взятия подстроки (`substr`) для `string_view` имеет сложность $O(1)$ — это просто сдвиг указателя и уменьшение размера. Никакой памяти не выделяется.

Подводные камни (Pitfalls)

С большой силой приходит большая ответственность. Поскольку `string_view` **не владеет** памятью, программист обязан следить за тем, чтобы исходная строка жила дольше, чем `string_view`.

Проблема 1: Dangling View (Висячая ссылка)

Классическая ошибка — создание `string_view` от временного объекта.

```
std::string_view get_greeting() {
    std::string s = "Hello, World!"; // Локальная переменная
```

```
    return s; // Неявное преобразование string -> string_view
} // s уничтожается, память освобождается

int main() {
    std::string_view sv = get_greeting();
    std::cout << sv; // UB: чтение освобожденной памяти (Use-after-free)
}
```

Также опасно комбинировать создание и использование в одной строке, если есть временные объекты:

```
std::string_view sv = std::string("Temporary") + " string";
// Временная строка умирает в конце выражения. sv смотрит в мусор.
```

Проблема 2: Отсутствие Null-Terminator

`std::string` гарантирует, что после данных следует байт `\0`, что позволяет использовать метод `.c_str()` для совместимости с C-функциями (`printf`, `fopen`).

`std::string_view` **не гарантирует** наличие нуль-терминатора, так как может указывать на середину другой строки.

```
std::string s = "Hello World";
std::string_view sv = s.substr(0, 5); // "Hello"

// ОШИБКА:
printf("%s", sv.data());
// sv.data() указывает на 'H'. printf будет печатать байты, пока не встретит \0.
// Он напечатает "Hello World" (в лучшем случае) или мусор за пределами s.
```

Если вам нужно передать `string_view` в функцию, ожидающую C-строку, вам придется сначала скопировать его в `std::string`:

```
std::string temp(sv);
legacy_function(temp.c_str());
```

Резюме раздела

1. **SSO** позволяет хранить короткие строки (обычно до 23 символов) без динамической аллокации. Это делает `std::string` очень эффективным для мелких данных.
2. Используйте `std::string_view` для передачи строк в функции и парсинга. Это избегает лишних копирований.
3. Передавайте `std::string_view` **по значению**.
4. Будьте предельно осторожны с временем жизни. **Никогда** не сохраняйте `string_view` на временные строки.
5. Помните, что `string_view` не является нуль-терминированной строкой.

Глава 6

Кейс-стади: Внутреннее устройство std::deque

Контейнер `std::deque` (Double-Ended Queue) — один из самых недооцененных и сложных контейнеров стандартной библиотеки. Часто его воспринимают просто как "вектор, в который можно быстро писать в начало". Однако его внутреннее устройство представляет собой уникальный компромисс между производительностью вектора и стабильностью ссылок связанного списка.

В этой главе мы разберем архитектуру `deque`, как она реализует "магию" расширения в обе стороны без копирования элементов и почему это важно для системного программирования.

Архитектура: Карта и Чанки

Наивная реализация двусторонней очереди могла бы быть кольцевым буфером на одном динамическом массиве. Но такая реализация имела бы проблему `std::vector`: при исчерпании емкости пришлось бы выделять новый огромный массив и переносить туда все элементы.

`std::deque` решает эту задачу иначе. Это **фрагментированный** контейнер.

> **Структура Deque**
>
> Логически `deque` — это бесконечный непрерывный массив. Физически — это набор фиксированных блоков памяти (**Chunks** или Buffers), управляемых центральным массивом указателей (**Map**).

Map (Карта)

Центральный элемент управления — это динамический массив указателей (`T** map`).

- Каждая ячейка карты указывает на отдельный блок памяти (чанк), хранящий элементы.
- Карта работает как кольцевой буфер: у нас есть индексы начала и конца последовательности заполненных блоков внутри карты.

Chunks (Блоки)

Сами данные хранятся в блоках фиксированного размера. Размер блока зависит от реализации (GCC, Clang, MSVC), но типичная эвристика — статический буфер размером около 512 байт (или минимум 16 элементов для мелких типов).

```
template <typename T>
class Deque {
private:
    // Параметры чанка
    static const size_t CHUNK_SIZE = 512 / sizeof(T);

    // Центральная "карта" - массив указателей на блоки
    T** map_;
    size_t map_size_;    // Размер самой карты (емкость указателей)

    // Итераторы начала и конца
    // Хранят индекс в карте + индекс внутри конкретного чанка
    iterator start_;
    iterator finish_;
};
```

Двухуровневая индексация

Доступ к произвольному элементу по индексу (`operator[]`) в `deque` сложнее, чем в векторе. Если в векторе это одна инструкция ассемблера (base + index * scale), то в деке это двухуровневая арифметика.

Чтобы получить элемент `deque[n]`, нужно:

1. Определить, в каком блоке находится элемент.
2. Определить смещение внутри этого блока.

Формула (упрощенно):

```
T& operator[](size_t n) {
    // 1. Учитываем смещение начала данных
    size_t pos = start_.cur_index + n;

    // 2. Вычисляем индекс блока в карте
    size_t node_index = pos / CHUNK_SIZE;

    // 3. Вычисляем смещение внутри блока
    size_t offset = pos % CHUNK_SIZE;

    // 4. Двойное разыменование: сначала карту, потом блок
    return map_[node_index][offset];
}
```

На заметку

Именно из-за двойного разыменования (double indirection) `std::deque` немного медленнее `std::vector` при последовательном проходе и случайном доступе. Однако он значительно быстрее `std::list`, так как данные в чанках лежат плотно, что обеспечивает хорошую локальность кэша (Cache Locality).

Главная особенность: Reference Stability

Киллер-фича `std::deque`, отличающая его от вектора — **стабильность ссылок**.

Рассмотрим сценарий:

```
std::vector<int> v = {1, 2, 3};
int& ref = v[0];
v.push_back(4); // Если capacity превышен, вектор реаллоцируется
// ref теперь указывает в освобожденную память (Dangling Reference)!
```

У вектора при расширении происходит выделение нового буфера и перемещение всех элементов. Все итераторы и ссылки инвалидируются.

У `deque` поведение принципиально иное:

1. При добавлении элементов (`push_back`/`push_front`) заполняются существующие чанки.
2. Если текущий чанк полон, выделяется **новый чанк** и добавляется в `map`.
3. Существующие элементы остаются на своих местах в старых чанках.

А что, если переполнилась сама карта (map)? В этот момент происходит реаллокация карты:

1. Выделяется новый массив указателей (обычно в 2 раза больше).
2. Указатели из старой карты копируются в центр новой.
3. Сами чанки данных **не трогаются**. Они остаются по тем же адресам в куче.

> **Важно!**
>
> Вставка в начало или конец `std::deque` **инвалидирует итераторы** (так как меняется внутренняя структура карты), но **НЕ инвалидирует ссылки и указатели** на элементы. Это критически важно, если вы храните указатели на элементы контейнера во внешних структурах.

Сценарии использования

Когда стоит выбрать `deque` вместо `vector`?

1. **Очередь сообщений:** Если вам нужно активно добавлять и удалять элементы с *обоих* концов. Вектор умеет быстро работать только с концом (`push_back`). Вставка в начало вектора — $O(N)$.

2. **Неинвалидация ссылок:** Если архитектура приложения требует, чтобы указатели на объекты сохранялись при росте контейнера.
3. **Фрагментация памяти:** Если вы храните миллионы объектов, вектор потребует найти непрерывный кусок памяти в сотни мегабайт, что может быть невозможно. Дек спокойно "съест" память кусками по 512 байт, разбросанными по всей куче.

Резюме раздела

- `std::deque` — это массив указателей на блоки фиксированного размера.
- Обеспечивает $O(1)$ вставку в начало и конец.
- Доступ по индексу чуть медленнее вектора (две инструкции чтения памяти).
- При реаллокации (расширении) элементы не перемещаются в памяти, ссылки остаются валидными.