

HSE

Faculty of Computer Science

Конспект углубленно- го курса по C++

AUTHOR

Your Name

COURSE

Advanced C++
Fall 2023

Оглавление

I Лекция 06 – Паттерны. ODR	3
1 Физическая структура программы: Компиляция, Линковка и ODR	4
1.1 Пайплайн сборки C++	4
1.1.1 1. Препроцессинг	4
1.1.2 2. Компиляция	4
1.1.3 3. Линковка (Компоновка)	5
1.2 One Definition Rule (ODR)	5
1.2.1 Declaration vs Definition	5
1.3 Проблема глобальных переменных в хедерах	6
1.4 Стратегии решения конфликтов линковки	6
1.4.1 1. extern (External Linkage)	6
1.4.2 2. static (Internal Linkage)	7
1.4.3 3. inline (Weak Linkage / COMDAT Folding)	7
1.5 Anonymous Namespaces	7
1.6 Библиотеки и LTO	8
1.6.1 Static (.a / .lib) vs Shared (.so / .dll)	8
1.6.2 Link Time Optimization (LTO)	8
1.7 Практические рекомендации	9
2 Управление зависимостями и идиома PImpl	10
2.1 Проблема транзитивных зависимостей	10
2.2 Реализация PImpl	11
2.2.1 Современный PImpl с std::unique_ptr	11
2.3 Проблема неполного типа (Incomplete Type) в деструкторе	13
2.4 ABI Stability (Бинарная совместимость)	13
2.5 Цена абстракции (Performance Trade-offs)	13
3 Архитектурные паттерны: Singleton и Template Method	15
3.1 Шаблонный метод (Template Method)	15
3.2 Паттерн Singleton (Одиночка)	16
3.2.1 Критика Singleton	17
3.3 Static Initialization Order Fiasco	17
3.4 Meyers Singleton	18
3.4.1 Thread Safety (Потокобезопасность)	18
3.5 Проблема разрушения (Dead Reference)	19
3.5.1 Leaky Singleton	19
3.6 Обобщение через CRTP	19
4 Глубокое погружение в исключения (Exception Safety)	22
4.1 Механика исключений: Zero-cost vs Stack Unwinding	22
4.1.1 Happy Path (Путь без ошибок)	22
4.1.2 Error Path (Путь ошибки)	22
4.2 Правила перехвата исключений	23
4.2.1 Slicing (Срезка) объектов	23
4.2.2 Порядок блоков catch	24

4.4 Гарантии безопасности исключений (Exception Safety Guarantees)	25
4.4.1 1. No-throw Guarantee (Гарантия отсутствия сбоев)	25
4.4.2 2. Strong Guarantee (Строгая гарантия)	25
4.4.3 3. Basic Guarantee (Базовая гарантия)	26
4.4.4 4. No Guarantee (Отсутствие гарантий)	26
4.5 Инварианты класса	26
4.6 Function-try-block	27

Часть I

Лекция 06 – Паттерны. ODR

Глава 1

Физическая структура программы: Компиляция, Линковка и ODR

Понимание того, как исходный код C++ превращается в исполняемый файл, является критическим навыком для системного программиста. В отличие от интерпретируемых языков или языков с JIT-компиляцией, C++ имеет строгую, многоступенчатую модель сборки, унаследованную от С. Ошибки на этапе линковки (*Unresolved external symbol*, *Multiple definition*) часто становятся следствием непонимания концепции единиц трансляции и правила одногого определения (ODR).

Пайплайн сборки C++

Процесс преобразования кода можно разделить на три изолированных этапа: препроцессинг, компиляция и линковка.

1. Препроцессинг

Препроцессор – это инструмент текстовой подстановки. Он ничего не знает о синтаксисе C++, типах данных или областях видимости.

- `#include "file.h"`: Содержимое файла `file.h` копируется байт-в-байт в место вызова директивы. Рекурсивные включения раскрываются полностью.
- `#define, #ifdef`: Обрабатываются макросы и условная компиляция.
- Комментарии удаляются.

Результатом работы препроцессора является *Translation Unit* (TU) – полный текст программы, готовый к компиляции. Если в исходном файле было 10 строк, а он включил заголовок на 10 000 строк, компилятор получит на вход 10 010 строк кода.

2. Компиляция

Компилятор работает с каждой единицей трансляции **изолированно**. При компиляции `a.cpp` компилятор не знает о существовании `b.cpp`.

1. **Лексический и синтаксический анализ**: Построение абстрактного синтаксического дерева (AST).
2. **Семантический анализ**: Проверка типов, перегрузок, шаблонов.

3. **Оптимизация:** Преобразование AST в промежуточное представление (IR) и применение оптимизаций (inlining, loop unrolling).
4. **Кодогенерация:** Создание объектного файла (.o или .obj).

Объектный файл содержит машинный код функций и данных, а также **таблицу символов** (Symbol Table). Символы делятся на:

- **Экспортируемые (Defined):** Функции и глобальные переменные, определенные в этом TU.
- **Импортируемые (Undefined):** Функции, которые были объявлены (declared), но не определены (defined) в текущем TU.

3. Линковка (Компоновка)

Линкер (Linker) собирает объектные файлы в единый исполняемый файл или библиотеку. Его задача — разрешить все Undefined символы, сопоставив их с Defined символами из других объектных файлов. Именно здесь проверяется правило ODR.

One Definition Rule (ODR)

Фундаментальное правило C++ (стандарт ISO/IEC 14882) гласит:

One Definition Rule (ODR)

1. В любой единице трансляции может быть не более одного определения любой переменной, функции, класса или шаблона.
2. Во всей программе должно быть **ровно одно** определение каждой используемой переменной или функции (non-inline).
3. Для классов и шаблонов разрешено иметь определения в нескольких TU, если они **побайтово идентичны** (token-for-token identical).

Нарушение пункта 2 приводит к ошибке линковки `multiple definition of symbol`. Нарушение пункта 3 (разные определения одного класса в разных файлах) приводит к неопределенному поведению (UB), которое крайне сложно диагностировать.

Declaration vs Definition

Важно различать объявление и определение:

- **Declaration (Объявление):** Сообщает компилятору о существовании сущности и её типе. Не выделяет память.

```

1 extern int x;           // Объявление переменной
2 void func();           // Объявление функции
3 class A;             // Forward declaration класса

```

- **Definition (Определение):** Создает сущность, выделяет память или генерирует код.

```

1 int x = 0;           // Определение (даже без инициализатора int x;)
2 void func() { }     // Определение (есть тело)
3 class A { int f; }; // Определение класса

```

Проблема глобальных переменных в хедерах

Рассмотрим классическую ошибку нарушения ODR.

header.h

```

1 #ifndef HEADER_H
2 #define HEADER_H
3 // ОШИБКА: Это определение глобальной переменной
4 int globalVar = 42;
5 #endif

```

a.cpp

```

1 #include "header.h"
2 // После препроцессинга здесь появится: int globalVar = 42;
3 // Компилятор создаст а.о с символом globalVar (Defined)

```

b.cpp

```

1 #include "header.h"
2 // После препроцессинга здесь тоже появится: int globalVar = 42;
3 // Компилятор создаст б.о с символом globalVar (Defined)

```

При попытке слинковать а.о и б.о линкер обнаружит два сильных символа с именем globalVar и выдаст ошибку `duplicate symbol`.

Стратегии решения конфликтов линковки

Существует три основных механизма для корректного разделения кода между единицами трансляции.

1. `extern` (External Linkage)

Традиционный подход С. В хедере мы только объявляем переменную, а определяем её ровно в одном .cpp файле.

```

1 // header.h
2 extern int globalVar; // Только объявление
3
4 // a.cpp
5 #include "header.h"
6 int globalVar = 42; // Определение (память выделена здесь)

```

```
7  
8 // b.cpp  
9 #include "header.h"  
10 // Использует символ globalVar, который будет найден в a.o
```

2. static (Internal Linkage)

Ключевое слово `static`, примененное к глобальной переменной или свободной функции, меняет тип линковки на *внутренний*.

```
1 // header.h  
2 static int globalVar = 42;
```

Если включить такой хедер в `a.cpp` и `b.cpp`:

- В `a.o` будет создана своя локальная копия `globalVar`.
- В `b.o` будет создана **другая**, независимая копия `globalVar`.

Линкер не увидит конфликта, так как символы не экспортируются наружу. Это решает ошибку сборки, но создает логическую ошибку, если вы ожидали, что переменная будет общей для всей программы (Singleton). Кроме того, это увеличивает размер бинарного файла.

3. inline (Weak Linkage / COMDAT Folding)

В C++17 появилась возможность объявлять переменные как `inline`. Это позволяет определять переменную в хедере.

```
1 // header.h  
2 inline int globalVar = 42; // C++17
```

Семантика `inline`: "Разрешено множественное определение". Компилятор помечает символ как слабый (weak). Линкер, встретив несколько копий такого символа, просто выбирает одну (обычно первую попавшуюся) и отбрасывает остальные. Все единицы трансляции будут ссылаться на один и тот же адрес памяти. Это идеальное решение для констант и глобальных настроек в современных стандартах.

Важно!

Шаблоны функций и классов неявно являются `inline`. Методы, определенные внутри тела класса, также неявно являются `inline`. Именно поэтому мы можем писать реализацию шаблонных классов полностью в `.h` файлах.

Anonymous Namespaces

В C++ вместо `static` для ограничения видимости внутри файла рекомендуется использовать анонимные пространства имен.

```

1 namespace {
2     int localHelper = 10;
3     void doSomethingInternal() { ... }
4 }
```

С точки зрения линковки это эквивалентно генерации уникального имени для пространства имен (например, namespace _unique_id_123), доступного только в текущем ТУ. Это предпочтительнее static, так как работает и для типов (классов), а не только для переменных и функций, позволяя передавать локальные типы в шаблоны.

Библиотеки и LTO

Static (.a / .lib) vs Shared (.so / .dll)

- **Static Library:** Это просто архив объектных файлов (аналог zip или tar с индексом). При линковке нужные .o файлы извлекаются из архива и копируются в исполняемый файл. Размер бинарника растет, но он становится самодостаточным.
- **Shared Library:** Не копируется в бинарник. Линкер лишь проверяет наличие символов и оставляет "заглушки". При запуске программы операционная система (Dynamic Loader) загружает библиотеку в память.
 - **Плюс:** Экономия оперативной памяти. Код библиотеки загружается в физическую память один раз и отображается в виртуальную память всех процессов, использующих её (механизм memory mapping).
 - **Минус:** Проблема зависимостей (DLL Hell), необходимость распространять библиотеки вместе с приложением.

Link Time Optimization (LTO)

Традиционная модель компиляции имеет недостаток: компилятор видит только один ТУ. Он не может заинлайнить функцию, определенную в другом .cpp файле, так как на этапе компиляции её тело недоступно.

LTO меняет этот процесс. При включении LTO (флаг -fLto в GCC/Clang) компилятор вместо машинного кода генерирует в объектных файлах промежуточное представление (Intermediate Representation, IR – например, GIMPLE или LLVM Bitcode).

Реальная генерация кода откладывается до этапа линковки. Линкер собирает IR со всех модулей, видит всю программу целиком и может выполнять агрессивные оптимизации:

- Cross-module inlining (инлайнинг функций между разными .cpp).
- Dead code elimination (удаление неиспользуемых функций, которые ранее считались экспортируемыми).

Цена LTO – значительное увеличение времени линковки и потребления памяти в процессе сборки.

Практические рекомендации

1. Используйте `#pragma once` для защиты хедеров. Это нестандартная директива, но она поддерживается всеми основными компиляторами и работает быстрее классических Include Guards (`#ifndef ...`), так как компилятору не нужно перечитывать файл.
2. Никогда не определяйте переменные в хедерах без `inline` или `constexpr`.
3. Шаблоны должны быть полностью определены в хедерах, так как компилятору нужно видеть их тело для инстанцирования конкретных типов.
4. Используйте анонимные пространства имен вместо `static` в .cpp файлах для сокрытия локальных помощников.

Глава 2

Управление зависимостями и идиома PImpl

Одной из фундаментальных проблем разработки крупных проектов на C++ является время компиляции. Из-за механизма включения заголовочных файлов (`#include`), изменение одной приватной переменной в низкоуровневом классе может вызвать каскадную перекомпиляцию сотен единиц трансляции, зависящих от этого заголовка.

Идиома `PImpl` (Pointer to Implementation), также известная как "Compilation Firewall" или "Opaque Pointer", — это архитектурный паттерн, призванный разорвать эти транзитивные зависимости и скрыть детали реализации от клиентского кода.

Проблема транзитивных зависимостей

В C++ определение класса в заголовочном файле должно быть полным. Компилятору необходимо знать точный размер объекта (значение `sizeof`), чтобы выделить память на стеке или внедрить объект в другой класс. Это означает, что все типы, используемые в качестве полей класса (даже приватных), должны быть полностью определены.

Рассмотрим пример класса `DatabaseConnection`, который использует тяжелую стороннюю библиотеку (например, драйвер PostgreSQL).

DatabaseConnection.h (Без PImpl)

```
1 #pragma once
2 // Мы вынуждены включить тяжелый хедер сторонней библиотеки,
3 // так как используем её типы в полях класса.
4 #include <libpq-fe.h>
5 #include <vector>
6 #include <string>
7
8 class DatabaseConnection {
9 public:
10     void connect(const std::string& connectionString);
11     void executeQuery(const std::string& query);
12
13 private:
14     // Детали реализации "протекают" в интерфейс.
15     // Любой файл, включающий DatabaseConnection.h,
```

```

16     // неявно включает <libpq-fe.h> и все его зависимости.
17     PGconn* pgConnection;
18     std::vector<std::string> cache;
19     int internalState;
20 };

```

Недостатки такого подхода:

- Header Bloat (Раздувание заголовков):** Клиентский код, которому нужен только метод connect, вынужден парсить тысячи строк кода из `<libpq-fe.h>`.
- Хрупкость сборки:** Если вы добавите новое приватное поле `int retryCount`, изменится `sizeof(DatabaseConnection)`. Все исходные файлы, использующие этот класс, обязаны быть перекомпилированы, иначе нарушится ODR (разные размеры объекта в разных TU).
- Загрязнение пространства имен:** Макросы и типы из сторонней библиотеки могут конфликтовать с кодом пользователя.

Реализация PImpl

Суть идиомы заключается в вынесении всех приватных полей в отдельную структуру (класс реализации), которая определяется только внутри .cpp файла. В публичном заголовке остается только указатель на эту структуру.

Поскольку размер указателя фиксирован и известен компилятору (обычно 8 байт на 64-битных системах), определение класса реализации в хедере не требуется. Достаточно предварительного объявления (forward declaration).

Современный PImpl с `std::unique_ptr`

В современном C++ для управления временем жизни реализации используется `std::unique_ptr`. Это обеспечивает RAII: реализация будет автоматически удалена вместе с публичным объектом.

DatabaseConnection.h (C PImpl)

```

1 #pragma once
2 #include <string>
3 #include <memory> // для std::unique_ptr
4
5 class DatabaseConnection {
6 public:
7     DatabaseConnection();
8     ~DatabaseConnection(); // Важно: объявляем, но не определяем здесь
9
10    // Перемещающие операции (Move Semantics) необходимы,
11    // так как unique_ptr нельзя копировать.
12    DatabaseConnection(DatabaseConnection&&) noexcept;
13    DatabaseConnection& operator=(DatabaseConnection&&) noexcept;
14
15    // Копирование запрещаем (или реализуем через глубокое копирование Impl)

```

```

16     DatabaseConnection(const DatabaseConnection&) = delete;
17     DatabaseConnection& operator=(const DatabaseConnection&) = delete;
18
19     void connect(const std::string& connectionString);
20     void executeQuery(const std::string& query);
21
22 private:
23     // Предварительное объявление.
24     // Класс Impl определен где-то в другом месте.
25     struct Impl;
26
27     // Указатель на реализацию.
28     // std::unique_ptr требует полного типа T только в момент вызова delete
29     std::unique_ptr<Impl> pImpl;
30 };

```

Обратите внимание: в этом файле нет #include <libpq-fe.h>. Зависимость полностью устранена.

DatabaseConnection.cpp

```

1 #include "DatabaseConnection.h"
2 // Включаем тяжелые зависимости ТОЛЬКО в файле реализации
3 #include <libpq-fe.h>
4 #include <vector>
5
6 // Полное определение класса реализации
7 struct DatabaseConnection::Impl {
8     PGconn* pgConnection = nullptr;
9     std::vector<std::string> cache;
10    int internalState = 0;
11
12    // Можно выносить сюда и приватные методы
13    void helperFunction() { /* ... */ }
14 };
15
16 // Конструктор: выделяем память под реализацию
17 DatabaseConnection::DatabaseConnection()
18     : pImpl(std::make_unique<Impl>()) {}
19
20 // Деструктор: именно здесь компилятору нужен полный тип Impl
21 DatabaseConnection::~DatabaseConnection() = default;
22
23 // Реализация перемещения
24 DatabaseConnection::DatabaseConnection(DatabaseConnection&&) noexcept = default;
25 DatabaseConnection& DatabaseConnection::operator=(DatabaseConnection&&)
26     noexcept = default;
27
28 // Проксирование вызовов
29 void DatabaseConnection::connect(const std::string& str) {
30     // Обращение к полям через стрелку pImpl->
31     pImpl->pgConnection = PQconnectdb(str.c_str());
32 }

```

Проблема неполного типа (Incomplete Type) в деструкторе

Распространенной ошибкой является попытка определить деструктор в заголовочном файле (или использовать деструктор по умолчанию, сгенерированный компилятором в хедере).

```

1 // DatabaseConnection.h
2 class DatabaseConnection {
3     // ОШИБКА, если Impl объявлен только предварительно (class Impl;)
4     ~DatabaseConnection() = default;
5     std::unique_ptr<Impl> pImpl;
6 };

```

Причина ошибки кроется в устройстве `std::unique_ptr`. В его деструкторе происходит вызов `delete ptr`. Оператор `delete` перед освобождением памяти вызывает деструктор объекта. Чтобы вызвать деструктор объекта `Impl`, компилятор должен видеть определение класса `Impl`. В заголовочном файле `Impl` – это неполный тип (incomplete type).

Использование `delete` на неполном типе в C++ является либо ошибкой компиляции, либо (в случае старых стандартов и сырых указателей) неопределенным поведением (UB). `std::unique_ptr` защищает от UB с помощью `static_assert`, проверяющего `sizeof(T) > 0`.

Важно!

Вы обязаны объявить деструктор в хедере, а реализовать его (даже через `= default`) в .cpp файле, **после** того как определена структура `Impl`.

ABI Stability (Бинарная совместимость)

Application Binary Interface (ABI) определяет, как объекты располагаются в памяти. В при мере без `PImpl` размер объекта `DatabaseConnection` зависит от реализации `std::vector` и внутренней структуры `PConn`.

Если вы обновите версию компилятора (что может изменить `sizeof(std::vector)`) или добавите приватное поле, размер класса изменится. Если ваше приложение динамически линкуется с библиотекой, предоставляющей этот класс, изменение размера приведет к смещению адресов полей и краху программы, если библиотека обновлена, а приложение – нет.

С `PImpl` размер класса `DatabaseConnection` всегда равен размеру одного `std::unique_ptr` (обычно размер машинного слова). Вы можете менять содержимое `Impl` как угодно: добавлять поля, менять контейнеры, удалять зависимости. Бинарный интерфейс публичного класса остается замороженным. Это критически важно для разработки библиотек (DLL/Shared Objects), которые должны обновляться без перекомпиляции клиентов.

Цена абстракции (Performance Trade-offs)

`PImpl` не является бесплатным. За изоляцию зависимостей приходится платить производительностью.

1. **Динамическая алокация:** При создании каждого объекта `DatabaseConnection` происходит обращение к куче (`new Impl`). Это медленнее, чем выделение памяти на стеке, и может вызвать фрагментацию памяти при создании миллионов мелких объектов.
2. **Indirection (Косвенная адресация):** Любой вызов метода требует разыменования указателя `pImpl->field`. Это дополнительная инструкция процессора и потенциальный промах кэша (Cache Miss), так как данные реализации лежат в куче, далеко от самого объекта.
3. **Размер кода:** Компилятор не может заинлайнить методы, реализация которых скрыта в `.cpp`.

Для решения проблемы с алокацией используется техника **Fast PImpl**. Вместо указателя используется буфер на стеке (`std::aligned_storage`), размер которого подбирается вручную.

```
1 // Fast PImpl (упрощенно)
2 class FastWidget {
3     // Резервируем память внутри объекта (на стеке)
4     // Риск: если Impl станет больше 64 байт, придется менять хедер
5     alignas(8) unsigned char storage[64];
6     struct Impl; // Реализация будет placement new в этот буфер
7 };
```

Этот подход возвращает скорость доступа и убирает алокацию, но возвращает зависимость от размера реализации (хотя и в меньшей степени – пока мы укладываемся в буфер, перекомпиляция не нужна).

Глава 3

Архитектурные паттерны: Singleton и Template Method

Паттерны проектирования (Design Patterns) – это не готовые библиотеки и не строгие алгоритмы, которые можно скопировать в код. Это формализованные описания способов решения часто встречающихся архитектурных задач. В контексте C++ паттерны приобретают специфику, связанную с управлением временем жизни объектов, полиморфизмом и возможностями метапрограммирования (шаблонов).

В этой главе мы рассмотрим два фундаментальных паттерна: *Template Method*, который позволяет выделять общий алгоритм в иерархиях классов, и *Singleton*, реализация которого в C++ прошла долгий путь эволюции от простых глобальных переменных до потокобезопасных конструкций на основе CRTP.

Шаблонный метод (Template Method)

Часто при проектировании иерархии классов возникает ситуация, когда общий алгоритм действий одинаков для всех наследников, но отдельные шаги этого алгоритма должны различаться.

В языках без множественного наследования реализации или без детерминированного разрушения объектов часто приходится дублировать логику. В C++ этот паттерн реализуется естественно через механизм виртуальных функций.

Template Method

Поведенческий паттерн, определяющий скелет алгоритма в базовом классе и delegирующий реализацию некоторых шагов подклассам. Подклассы могут переопределять шаги алгоритма, не меняя его общей структуры.

Рассмотрим пример симуляции жизни животных. У всех животных день проходит по одному сценарию: проснуться, издать звук, поесть. Процесс пробуждения и приема пищи (в нашей упрощенной модели) одинаков, а вот звуки – уникальны.

```
1 #include <iostream>
2 #include <string>
3
4 // Базовый класс определяет алгоритм
```

```

5  class Animal {
6  public:
7      virtual ~Animal() = default;
8
9      // Шаблонный метод.
10     // Он НЕ виртуальный, чтобы зафиксировать структуру алгоритма.
11     void liveOneDay() {
12         wakeUp();
13         makeSound(); // Точка кастомизации (Virtual Hook)
14         eat();
15     }
16
17 protected:
18     // Общие шаги реализации
19     void wakeUp() const {
20         std::cout << "Opening eyes..." << std::endl;
21     }
22
23     void eat() const {
24         std::cout << "Eating food..." << std::endl;
25     }
26
27     // Чисто виртуальный метод – обязан быть реализован в наследнике
28     virtual void makeSound() const = 0;
29 };
30
31 // Конкретная реализация
32 class Dog : public Animal {
33 protected:
34     void makeSound() const override {
35         std::cout << "Woof! Woof!" << std::endl;
36     }
37 };
38
39 class Cat : public Animal {
40 protected:
41     void makeSound() const override {
42         std::cout << "Meow..." << std::endl;
43     }
44 };

```

В идиоматике C++ этот подход часто пересекается с идиомой **NVI (Non-Virtual Interface)**. NVI рекомендует делать публичные методы невиртуальными (обеспечивая стабильный интерфейс и проверки инвариантов), а точки расширения (виртуальные методы) делать `protected` или `private`.

Это позволяет базовому классу `Animal` гарантировать, что звук будет издан именно между пробуждением и едой, и никакой наследник не сможет случайно сломать этот порядок, забыв вызвать базовый метод.

Паттерн Singleton (Одиночка)

`Singleton` – один из самых спорных, но часто используемых паттернов. Его задача:

1. Гарантировать, что у класса есть только один экземпляр.
2. Предоставить к нему глобальную точку доступа.

Примеры использования:

- **Аллокатор памяти:** Система управления памятью обычно глобальна для процесса.
- **Файловая система / Логгер:** Централизованная запись логов в один файл требует синхронизации, которую проще обеспечить через единый объект.
- **Драйвер видеокарты:** Физическое устройство одно, и доступ к нему должен быть монопольным.

Критика Singleton

Singleton часто называют "антипаттерном" из-за проблем, которые он привносит:

- **Скрытые зависимости:** Если функция принимает аргументы, её зависимости явны. Если функция внутри себя вызывает `Logger::getInstance()`, зависимость скрыта.
- **Проблемы тестирования:** Невозможно изолировать тесты. Если один тест изменил состояние синглтона, второй тест может упасть. Нельзя запустить два экземпляра "приложения" параллельно в одном тестовом процессе.
- **Глобальное состояние:** Усложняет понимание потока данных и многопоточности.

Тем не менее, в системном программировании Singleton остается необходимым инструментом.

Static Initialization Order Fiasco

Почему нельзя просто создать глобальную переменную?

```

1 // Logger.h
2 class Logger { ... };
3 extern Logger globalLogger;
4
5 // Logger.cpp
6 Logger globalLogger;

```

Проблема возникает, когда у нас есть **несколько** глобальных статических объектов в **разных** единицах трансляции (файлах .cpp), и один из них использует другой в своем конструкторе.

Представим ситуацию:

1. В `Logger.cpp` определен `globalLogger`.
2. В `Application.cpp` определен объект `Application app`, который в конструкторе пишет лог: `globalLogger.log("App started")`.

Стандарт C++ **не определяет** порядок инициализации глобальных переменных, находящихся в разных единицах трансляции. Линкер может собрать их в любом порядке.

Важно!

Если app будет инициализировано раньше, чем `globalLogger`, конструктор `Application` обратится к неинициализированной памяти (в лучшем случае заполненной нулями). Это приведет к краху программы (Segmentation Fault) или неопределенному поведению до начала функции `main`.

Это явление называется *Static Initialization Order Fiasco*.

Meyers Singleton

Решением проблемы инициализации является идиома, популяризированная Скоттом Мейерсом. Идея заключается в переносе статической переменной из глобальной области видимости внутрь функции.

```

1  class Logger {
2  private:
3      Logger() { /* ... */ }
4      // Запрещаем копирование
5      Logger(const Logger&) = delete;
6      Logger& operator=(const Logger&) = delete;
7
8  public:
9      static Logger& getInstance() {
10         // Статическая локальная переменная.
11         // Инициализируется ПРИ ПЕРВОМ вызове функции.
12         static Logger instance;
13         return instance;
14     }
15
16     void log(const std::string& msg) { /* ... */ }
17 };

```

Теперь порядок инициализации детерминирован. Если конструктор `Application` вызовет `getInstance()`, логгер будет создан именно в этот момент.

Thread Safety (Потокобезопасность)

До стандарта C++11 инициализация статических локальных переменных не была потокобезопасной. Если два потока одновременно заходили в `getInstance()` в первый раз, мог возникнуть race condition (двойное создание). Приходилось использовать паттерн *Double-Checked Locking*.

В C++11 и новее стандарт гарантирует: **инициализация статических локальных переменных потокобезопасна**. Компилятор автоматически генерирует блокировки (обычно через атомарные флаги или мьютексы), чтобы только один поток выполнил инициализацию. Это называют "Magic Statics".

Проблема разрушения (Dead Reference)

Meyers Singleton решает проблему инициализации, но оставляет проблему разрушения. Статические объекты разрушаются в порядке, обратном созданию, после выхода из `main`.

Сценарий катастрофы:

1. `main` завершается.
2. Разрушается `Logger` (так как он был создан последним, лениво).
3. Разрушается глобальный объект `Application`.
4. В деструкторе `Application` происходит попытка записать прощальное сообщение: `Logger::getInstance().log("App stopped")`.
5. Метод `getInstance` возвращает ссылку на уже уничтоженный объект `Logger`.
6. Use-after-free, UB, крах.

Leaky Singleton

Чтобы избежать этой проблемы, в системном программировании часто используют "утекающий" синглтон. Мы создаем объект через `new`, но никогда не вызываем `delete`.

```

1 static Logger& getInstance() {
2     // Память выделяется в куче и никогда не освобождается.
3     static Logger* instance = new Logger();
4     return *instance;
5 }
```

С точки зрения C++, это утечка памяти (Memory Leak). Однако, это безопасная утечка. Когда процесс завершается, операционная система всё равно реиспользует все страницы памяти, принадлежавшие процессу. Зато объект `Logger` гарантированно живёт до самого последнего момента существования процесса, и любой деструктор может безопасно им воспользоваться.

Обобщение через CRTP

В крупном проекте писать метод `getInstance` и закрывать конструкторы для каждого синглтона утомительно. Мы можем захотеть создать базовый класс `Singleton<T>`, который инкапсулирует эту логику.

Однако, `Singleton` должен знать тип создаваемого класса, а создаваемый класс должен наследоваться от `Singleton`. Возникает циклическая зависимость, которая разрешается с помощью идиомы **CRTP** (**C**uriously **R**ecurring **T**emplate **P**attern) – Странно Рекурсивный Шаблон.

```

1 #include <utility> // для std::forward
2
3 template <typename T>
4 class Singleton {
5 public:
```

```

6   // Variadic templates позволяют передать аргументы в конструктор T
7   template <typename ... Args>
8     static T& getInstance(Args&&... args) {
9       // Создаем через new (Leaky Singleton), чтобы избежать проблем с
10      // деструкцией
11      static T* instance = new T(std::forward<Args>(args)...);
12      return *instance;
13    }
14
15  protected:
16    Singleton() = default;
17    ~Singleton() = default;
18
19  public:
20    // Запрещаем копирование и перемещение для самого Singleton
21    Singleton(const Singleton&) = delete;
22    Singleton& operator=(const Singleton&) = delete;
23
24  // Пример использования
25  class FileSystem : public Singleton<FileSystem> {
26    // Важно: Singleton<FileSystem> должен иметь доступ к приватному конструктору
27    friend class Singleton<FileSystem>;
28
29  private:
30    FileSystem(const std::string& root) {
31      std::cout << "FS mounted at " << root << "\n";
32    }
33
34  public:
35    void readFile(const std::string& path) { /* ... */ }
36  };
37
38  int main() {
39    // Первый вызов инициализирует объект
40    FileSystem::getInstance("/var/data").readFile("config.txt");
41
42    // Последующие вызовы возвращают тот же объект (аргументы игнорируются)
43    FileSystem::getInstance().readFile("log.txt");
44  }

```

Разбор механизма CRTP:

1. **class FileSystem : public Singleton<FileSystem>**: В момент объявления класса `FileSystem`, шаблон `Singleton` инстанцируется типом `FileSystem`. Хотя `FileSystem` еще является неполным типом (incomplete type), его можно использовать как аргумент шаблона, если шаблон не обращается к внутренностям типа в момент объявления.
2. Метод `getInstance` является статическим членом базового класса, но возвращает ссылку на `T` (то есть на наследника).
3. **friend class Singleton<FileSystem>**: Это ключевой момент. Так как конструктор `FileSystem` приватен (чтобы никто не создал второй экземпляр), базовый класс не мог бы вызвать `new T(...)`. Дружба дает базовому классу права на доступ к приватным членам наследника.

Этот паттерн позволяет элегантно вынести инфраструктурную логику (управление единственностью) из бизнес-логики класса.

Глава 4

Глубокое погружение в исключения (Exception Safety)

Механизм исключений в C++ – это мощный инструмент обработки ошибок, который часто понимается поверхностно. В отличие от кодов возврата (как в C или Go) или монад (как `Result<T, E>` в Rust), исключения в C++ предлагают автоматическую раскрутку стека и нелокальную передачу управления. Однако неправильное использование исключений ведет к утечкам ресурсов, неопределенному поведению и деградации производительности.

Эта глава посвящена не синтаксису `try-catch`, а внутренней механике процесса (*Under the hood*), гарантиям безопасности (Exception Safety Guarantees) и правильному написанию устойчивого кода.

Механика исключений: Zero-cost vs Stack Unwinding

Часто можно услышать, что "исключения медленные". Это утверждение верно лишь отчасти. Современные компиляторы (GCC, Clang, MSVC) реализуют модель, называемую **Zero-cost exceptions** (на Itanium C++ ABI).

Happy Path (Путь без ошибок)

Пока исключение **не** брошено, накладные расходы на поддержку механизма исключений практически равны нулю. В коде нет проверок `if (error)` после каждого вызова функции. Процессор выполняет линейный поток инструкций, что благоприятно для предсказателя переходов (Branch Predictor) и кэша инструкций.

Error Path (Путь ошибки)

Когда выполняется оператор `throw`, происходит дорогая операция, называемая **Stack Unwinding** (Раскрутка стека):

1. **Поиск обработчика:** Runtime-система (например, `libunwind`) просматривает таблицу исключений (`.eh_frame` или аналогичную секцию), сгенерированную компилятором. Эта таблица сопоставляет диапазоны адресов инструкций (Program Counter) с информацией о том, какие деструкторы нужно вызывать в текущем кадре стека и есть ли здесь блок `catch`.

2. **Вызов деструкторов:** Если в текущей функции нет подходящего `catch`, система вызывает деструкторы всех локальных объектов, созданных с начала функции до точки выброса.
3. **Раскрутка:** Стек урезается (stack pointer смещается), управление передается вызывающей функции, и процесс повторяется (шаг 1), пока не будет найден `catch` или пока не будет достигнут `main`.
4. **Terminate:** Если стек раскрутился до `main`, а обработчик не найден, вызывается `std::terminate` (обычно `abort()`).

Этот процесс включает парсинг таблиц, множество косвенных переходов и вызовов, что действительно очень медленно (тысячи тактов процессора). Поэтому исключения не должны использоваться для управления потоком выполнения в штатном режиме (например, для выхода из цикла).

Правила перехвата исключений

Slicing (Срезка) объектов

Ловить исключения нужно **строго по ссылке** (желательно константной). Ловля по значению приводит к срезке полиморфной части объекта.

```

1 #include <iostream>
2 #include <exception>
3
4 struct MyError : std::exception {
5     const char* what() const noexcept override {
6         return "My Custom Error";
7     }
8     int errorCode = 500;
9 };
10
11 void badHandler() {
12     try {
13         throw MyError();
14     }
15     // ПЛОХО: catch по значению.
16     // Создается копия std::exception, отбрасывая часть MyError.
17     catch (std::exception e) {
18         // e.what() вызовет метод базового класса std::exception,
19         // а не переопределенный MyError::what()!
20         std::cout << e.what() << "\n";
21         // e.errorCode недоступен
22     }
23 }
24
25 void goodHandler() {
26     try {
27         throw MyError();
28     }
29     // ХОРОШО: catch по ссылке. Полиморфизм работает.
30     catch (const std::exception& e) {

```

```

31         std::cout << e.what() << "\n"; // Выведет "My Custom Error"
32     }
33 }
```

Порядок блоков catch

Блоки `catch` проверяются сверху вниз. Компилятор не всегда предупредит вас, если вы попытаетесь поймать базовый тип раньше производного.

```

1 try {
2     // ...
3 }
4 catch (const std::exception& e) {
5     // Сюда попадут ВСЕ наследники std::exception
6 }
7 catch (const std::runtime_error& e) {
8     // Мертвый код (Dead Code)!
9     // runtime_error наследуется от exception,
10    // поэтому он будет перехвачен первым блоком.
11 }
```

Спецификатор noexcept

Ключевое слово `noexcept` – это часть контракта функции.

- `void func() noexcept;` – гарантирует, что функция не выбросит исключение.
- Если исключение всё же вылетит из `noexcept`-функции, программа немедленно вызовет `std::terminate()`. Раскрутка стека может не произойти (деструкторы не вызовутся).

Компилятор использует эту информацию для агрессивных оптимизаций. Если функция помечена `noexcept`, компилятору не нужно генерировать таблицы раскрутки стека для вызовов внутри неё.

Вектор и noexcept move-конструкторы

Это критически важный аспект для производительности контейнеров. Рассмотрим `std::vector`: Если вектору не хватает места (`capacity`), он должен:

1. Выделить новый блок памяти.
2. Перенести элементы из старого блока в новый.
3. Удалить старый блок.

При переносе элементов вектор предпочитает использовать *перемещение* (*move constructor*), так как это дешево (копирование указателей). Однако, если *move*-конструктор элемента может выбросить исключение, возникает проблема нарушения транзакционности.

Представьте, что мы перенесли 5 элементов из 10, и 6-й элемент выбросил исключение при перемещении.

- Старый вектор испорчен (из 5 элементов ресурсы украдены).
- Новый вектор недостроен.
- Восстановить исходное состояние невозможно.

Поэтому `std::vector` использует идиому `std::move_if_noexcept`:

```

1 // Псевдокод логики вектора при реаллокации
2 if (std::is_nothrow_move_constructible<T>::value) {
3     // Безопасно муваем. Если упадем - значит terminate, терять нечего.
4     move_elements();
5 } else {
6     // Приходится копировать.
7     // Если копия упадет, старый вектор останется нетронутым.
8     copy_elements();
9 }
```

Важно!

Всегда помечайте перемещающие конструкторы и операторы присваивания как `noexcept`, иначе стандартные контейнеры деградируют до копирования, что убьет производительность.

Гарантии безопасности исключений (Exception Safety Guarantees)

При проектировании классов вы должны документировать, что произойдет с объектом, если его метод выбросит исключение. Абстракция не должна "течь" или оставлять систему в рассогласованном состоянии.

Выделяют четыре уровня гарантий (от слабой к сильной):

1. No-throw Guarantee (Гарантия отсутствия сбоев)

Функция никогда не выбрасывает исключений. Это обязательное требование для:

- **Деструкторов:** Выброс исключения из деструктора во время раскрутки стека (когда уже летит другое исключение) приводит к `std::terminate`.
- **Функций освобождения памяти** (`delete`, `free`).
- **Swap:** Функция обмена должна быть надежной, так как на ней строятся другие гарантии.

2. Strong Guarantee (Строгая гарантия)

Транзакционная семантика: "Commit or Rollback". Если функция завершилась с исключением, состояние программы остается таким же, каким было до вызова функции. Никаких побочных эффектов.

Пример реализации через идиому **Copy-and-Swap**:

```

1 class Widget {
2   private:
3     std::vector<int> data;
4   public:
5     // Мы хотим заменить данные, но безопасно.
6     void updateData(const std::vector<int>& newData) {
7       // 1. Создаем временную копию (может бросить исключение).
8       // Если бросит здесь - this не изменится.
9       std::vector<int> temp = newData;
10
11      // 2. Меняем состояние через небросающий swap.
12      // std::vector::swap - noexcept.
13      using std::swap;
14      swap(this->data, temp);
15
16      // 3. temp (со старыми данными) разрушается при выходе.
17    }
18 };

```

3. Basic Guarantee (Базовая гарантия)

Если произошло исключение:

- Нет утечек ресурсов (память, файлы, мьютексы освобождены).
- Инварианты класса сохранены (внутреннее состояние согласовано).
- Однако данные могут быть изменены и непредсказуемы (но валидны).

Это минимальный уровень, требуемый от корректного C++ кода. Обычно обеспечивается использованием RAII (`std::unique_ptr`, `std::lock_guard`).

4. No Guarantee (Отсутствие гарантий)

Если произошло исключение, всё сломалось: память утекла, данные коррумпированы. Это недопустимо в качественном коде.

Инварианты класса

Инвариант – это логическое условие, которое всегда истинно для валидного объекта (кроме момента выполнения методов, меняющих состояние).

Пример класса, уязвимого к исключениям:

```

1 class Buffer {
2   int* ptr = nullptr;
3   size_t size = 0;
4   public:
5     Buffer(size_t n) {
6       // ОПАСНО!
7       size = n;           // 1. Установили размер
8       ptr = new int[n]; // 2. Пытаемся выделить память

```

```

9     // Если new бросит std::bad_alloc:
10    // - Конструктор прервется.
11    // - Деструктор НЕ будет вызван (объект не достроен).
12    // - У нас остался "объект" с size > 0, но без памяти.
13    // Хотя самого объекта технически нет, это нарушение логики построения.
14 }
15 };

```

Более тонкий пример нарушения инварианта (Basic Guarantee):

```

1 void process() {
2     size_t oldSize = size;
3     size = 0;          // Ломаем инвариант "size соответствует данным"
4     doWork();         // Если тут throw...
5     size = oldSize;  // ...восстановление не выполнится
6     // Объект остался с size=0, но старыми данными.
7 }

```

Решение: использовать ScopeGuard или менять состояние в самом конце (после всех опасных операций).

Function-try-block

Специфическая конструкция C++, позволяющая перехватить исключения, возникающие в **списке инициализации конструктора** (Member Initializer List). Это единственный способ обработать ошибку в конструкторе базового класса или члена класса.

```

1 class Service {
2     Resource res; // Конструктор Resource может бросить
3 public:
4     Service() try : res(42) {
5         // Тело конструктора
6     }
7     catch (const std::exception& e) {
8         // Мы попадем сюда, если res(42) упадет.
9         // ВАЖНО: Исключение здесь НЕЛЬЗЯ подавить.
10        // Даже если мы не напишем throw;, оно будет перевыброшено автоматически.
11        // Смысл блока только в логировании или трансляции исключения.
12        std::cerr << "Init failed: " << e.what() << "\n";
13    }
14 };

```

Использовать function-try-block для обычных функций не рекомендуется (это эквивалентно try внутри всего тела), а в конструкторах он нужен крайне редко, так как обычно лучше дать исключению улететь наверх, чем пытаться починить недостроенный объект.