HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 07 – Метапрограммирование

Глава 1

Препроцессор и Макромагия: От текстовой подстановки до кодогенерации

Метапрограммирование в C++ часто ассоциируется исключительно с шаблонами (templates), однако исторически первым и до сих пор широко используемым инструментом метапрограммирования является препроцессор.

Препроцессор языка C (и C++) — это утилита, которая обрабатывает исходный код **до** этапа компиляции. Он не знает синтаксиса C++, не понимает типов данных, областей видимости или классов. Для препроцессора ваш код — это просто поток текста (набор токенов), над которым производятся операции подстановки, склейки и условного включения.

Фундаментальные механики

Работа препроцессора управляется директивами, начинающимися с символа #. Эти директивы выполняются на самом раннем этапе трансляции, превращая исходный файл `.cpp` в единицу трансляции (translation unit), готовую к синтаксическому анализу компилятором.

Include Guards и условная компиляция

Одной из базовых задач препроцессора является управление зависимостями и платформозависимым кодом. Директива `#include` буквально копирует содержимое указанного файла в точку вызова. Это создает проблему множественного включения: если заголовочный файл `A.h` включен в `B.h` и `C.h`, а `main.cpp` включает и `B`, и `C`, то содержимое `A.h` попадет в `main.cpp` дважды, что приведет к ошибкам переопределения символов (ODR — One Definition Rule).

Традиционное решение — Include Guards:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

struct MyStruct { /* ... */ };

#endif // MY_HEADER_H
```

Также препроцессор используется для проверки окружения. Макросы позволяют включать или исключать куски кода в зависимости от ОС, компилятора или конфигурации сбор-

ки (Debug/Release).

```
#ifdef _WIN32
    #include <Windows.h>
#elif defined(__unix__)
    #include <unistd.h>
#else
    #error "Unknown platform"
#endif
```

> **Важно!**
>
> Директива `#error` прерывает компиляцию с указанным сообщением. Это полезно для гарантии того, что код не будет собран на неподдерживаемой архитектуре, вместо того чтобы получить тысячи непонятных синтаксических ошибок далее.

Предопределенные макросы

Компилятор предоставляет набор стандартных макросов, которые содержат информацию о текущем контексте компиляции. Они часто используются для логирования и отладки.

- `__LINE__` (int): Текущий номер строки в исходном файле.
- `__FILE__` (string literal): Имя текущего файла.
- `__func__` (string literal): Имя текущей функции (стандартизировано в C99/C++11). В GCC/Clang также доступен нестандартный `__PRETTY_FUNCTION__`, который выводит полную сигнатуру функции, включая типы аргументов и шаблонов.

Имена, начинающиеся с двух подчеркиваний (`__`) или с одного подчеркивания и заглавной буквы (`_Big`), зарезервированы стандартом для реализации. Определение собственных макросов с такими именами является Undefined Behavior.

Макросы как функции

Функциональные макросы позволяют выполнять текстовую подстановку аргументов. В отличие от шаблонов или `inline`-функций, макросы не проверяют типы и вычисляются путем простой замены токенов.

Опасность приоритета операций

При написании макросов критически важно оборачивать каждый аргумент и все выражение целиком в скобки. Поскольку препроцессор просто вставляет текст, операторы с низким приоритетом могут "захватить" соседние выражения.

Рассмотрим некорректный макрос:

```
#define MUL(a, b) a * b

int res = MUL(2 + 3, 4 + 5);
// Раскрывается в: 2 + 3 * 4 + 5
// Результат: 2 + 12 + 5 = 19
```

```
// Ожидалось: 5 * 9 = 45
```

Корректная реализация:

```
#define MUL(a, b) ((a) * (b))
```

Stringification (Оператор #)

Оператор решетки # используется для превращения аргумента макроса в строковый литерал. Это невозможно сделать средствами самого C++, так как имена переменных теряются после компиляции.

Это ключевой механизм для реализации `assert`-ов, которые выводят само проверяемое выражение при ошибке.

```
#include <iostream>
#include <cstdlib>

#define MY_ASSERT(expr) \
    if (!(expr)) { \
        std::cerr << "Assertion failed: " << #expr \
                  << ", file " << __FILE__ \
                  << ", line " << __LINE__ << std::endl; \
        std::abort(); \
    }

int main() {
    int x = 5;
    // Вывод: Assertion failed: x == 2, file main.cpp, line 15
    MY_ASSERT(x == 2);
}
```

Обратите внимание: препроцессор видит выражение `x == 2` как последовательность токенов, а оператор # превращает их в строку `"x == 2"`.

Идиома do-while(0)

При создании многострочных макросов возникает проблема корректного синтаксиса в управляющих конструкциях. Если просто обернуть код в фигурные скобки `{ }`, использование макроса в ветке `if` может сломать `else`.

Пример проблемы:

```
#define LOG_ERROR(msg) \
    { std::cerr << "Error: "; std::cerr << msg << std::endl; }

if (cond)
    LOG_ERROR("Fail"); // Точка с запятой здесь завершает if!
else                   // ОШИБКА: else без if
    /* ... */
```

Раскрытие кода:

```
if (cond)
    { std::cerr << "Error: "; std::cerr << "Fail" << std::endl; };
else // Синтаксическая ошибка: лишняя ; перед else
```

Решение — использование цикла `do { ... } while(0)`. Эта конструкция требует точки с запятой на конце, что делает вызов макроса синтаксически идентичным вызову обычной функции.

```
#define LOG_ERROR(msg) \
    do { \
        std::cerr << "Error: "; \
        std::cerr << msg << std::endl; \
    } while(0)
```

Variadic Macros и перегрузка

C99 и C++11 принесли поддержку макросов с переменным числом аргументов. Они обозначаются многоточием `...`, а доступ к аргументам осуществляется через идентификатор `__VA_ARGS__`.

Одной из самых мощных (и неочевидных) техник является перегрузка макросов по количеству аргументов. Поскольку препроцессор не поддерживает перегрузку функций напрямую, приходится использовать трюк с подменой макроса-диспетчера.

Механизм выбора N-го аргумента

Задача: реализовать макрос `ENSURE`, который можно вызывать как `ENSURE(cond)` или `ENSURE(cond, message)`.

Алгоритм: 1. Создать два макроса реализации: `ENSURE_1(cond)` и `ENSURE_2(cond, msg)`. 2. Создать макрос-селектор, который выбирает нужную реализацию в зависимости от количества переданных аргументов.

```
// Реализации
#define ENSURE_1(cond) \
    if (!(cond)) { std::cerr << "Check failed: " << #cond << std::endl;
    ↪ std::abort(); }

#define ENSURE_2(cond, msg) \
    if (!(cond)) { std::cerr << msg << std::endl; std::abort(); }

// Магия выбора
// Этот макрос принимает набор аргументов и возвращает 3-й аргумент
#define GET_3RD_ARG(arg1, arg2, arg3, ...) arg3

// Макрос-диспетчер
// Если передано 1 аргумент (x):
//   GET_3RD_ARG(x, ENSURE_2, ENSURE_1) -> вернет ENSURE_1
```

```
// Если передано 2 аргумента (x, y):
//   GET_3RD_ARG(x, y, ENSURE_2, ENSURE_1) -> вернет ENSURE_2
#define ENSURE_MACRO_CHOOSER(...) \
    GET_3RD_ARG(__VA_ARGS__, ENSURE_2, ENSURE_1)

// Итоговый макрос
#define ENSURE(...) \
    ENSURE_MACRO_CHOOSER(__VA_ARGS__)(__VA_ARGS__)
```

Этот механизм работает за счет сдвига аргументов. Макрос `GET_3RD_ARG` всегда возвращает третий элемент из списка. Подставляя в него пользовательские аргументы `__VA_ARGS__` в начало, мы сдвигаем служебные имена макросов (`ENSURE_2`, `ENSURE_1`) на нужную позицию.

X-Macros: Кодогенерация списков

Одной из самых частых проблем C++ является отсутствие рефлексии. Например, при объявлении `enum` для цветов, мы теряем текстовое представление имени цвета. Чтобы вывести имя цвета в лог, приходится писать функцию `ToString` с большим `switch-case`, дублируя имена констант.

Техника **X-Macros** позволяет определить данные один раз и генерировать различный код (enum, массив строк, switch-case) путем многократного включения списка.

Реализация X-Macro

Идея состоит в том, чтобы список элементов был определен как последовательность вызовов некоторого (пока не определенного) макроса `X`.

```
// colors.def (или просто #define внутри файла)
#define LIST_OF_COLORS(X) \
    X(Red) \
    X(Green) \
    X(Blue) \
    X(Yellow)
```

Теперь мы можем использовать этот список для генерации кода.

Шаг 1: Генерация enum Мы определяем макрос `X(name)` так, чтобы он разворачивался в `name,`.

```
enum class Color {
    #define X(name) name,
    LIST_OF_COLORS(X)
    #undef X
};
// Раскроется в:
// enum class Color {
//     Red,
//     Green,
//     Blue,
```

```
// Yellow,
// };
```

Шаг 2: Генерация функции ToString Мы переопределяем `X(name)` так, чтобы он генерировал `case`.

```
const char* ToString(Color c) {
    switch (c) {
        #define X(name) case Color::name: return #name;
        LIST_OF_COLORS(X)
        #undef X
    }
    return "Unknown";
}
// Раскроется в:
// switch (c) {
//     case Color::Red: return "Red";
//     case Color::Green: return "Green";
//     ...
// }
```

> **X-Macros**
>
> Это паттерн использования препроцессора, при котором данные отделяются от их представления. Данные описываются в виде списка вызовов макроса-заглушки, который переопределяется контекстно для генерации различных структур кода (объявлений, массивов, операторов switch).

Эта техника широко используется в крупных проектах (например, в LLVM или ClickHouse) для синхронизации конфигураций, кодов ошибок или настроек CLI, гарантируя, что добавление нового элемента в список автоматически обновит все связанные структуры данных.

Резюме раздела

- Препроцессор — мощный инструмент текстовой генерации, работающий до компилятора.
- Всегда используйте скобки вокруг аргументов макроса и идиому `do-while(0)` для многострочных макросов.
- Оператор # позволяет получить строковое представление кода (полезно для отладки и сериализации).
- X-Macros позволяют эмулировать рефлексию и избегать дублирования кода при работе с перечислениями и списками свойств.

Глава 2

Метапрограммирование на Шаблонах (TMP) и Constexpr

Метапрограммирование в C++ — это методика написания программ, которые выполняются компилятором и генерируют другие программы (или константы) в качестве своего вывода. Это Turing-complete подсистема языка, встроенная непосредственно в процесс компиляции.

Исторически метапрограммирование в C++ развивалось от случайного открытия возможности вычислений на шаблонах (Template Metaprogramming — TMP) до полноценной поддержки вычислений времени компиляции через механизм `constexpr`. В этой главе мы рассмотрим эволюцию этих подходов: от сложных рекурсивных структур к современному императивному коду, выполняемому на этапе трансляции.

Классический TMP: Шаблоны как функциональный язык

До стандарта C++11 шаблоны были единственным способом заставить компилятор выполнять произвольные вычисления. Этот подход (Legacy TMP) базируется на функциональной парадигме. В нем отсутствуют переменные (все данные иммутабельны) и циклы.

> **Template Metaprogramming (TMP)**
>
> Парадигма, где:
>
> - **Функции** представлены шаблонными структурами (`struct` или `class`).
> - **Аргументы** передаются как параметры шаблона (`<int N, typename T>`).
> - **Возвращаемые значения** — это вложенные константы (`static const value`) или определения типов (`typedef type`).
> - **Циклы** реализуются через рекурсию.
> - **Ветвление** реализуется через специализацию шаблонов.

Рекурсия и Специализация: Вычисление факториала

Классическим примером ("Hello World" от мира TMP) является вычисление факториала. Поскольку циклы `for` или `while` недоступны на уровне инстанцирования шаблонов, используется рекурсивное определение.

```cpp
#include <iostream>

// Общий шаблон: N! = N * (N-1)!
template <unsigned long N>
struct Factorial {
    // "Возвращаемое значение" вычисляется рекурсивно
    static const unsigned long value = N * Factorial<N - 1>::value;
};

// Условие выхода из рекурсии: специализация для 0
// 0! = 1
template <>
struct Factorial<0> {
    static const unsigned long value = 1;
};

int main() {
    // Вычисление происходит полностью во время компиляции.
    // В бинарный код попадает только константа 3628800.
    std::cout << Factorial<10>::value << std::endl;
    return 0;
}
```

Механика работы: Когда компилятор встречает `Factorial<10>`, он пытается инстанцировать эту структуру. Внутри он видит обращение к `Factorial<9>`. Это запускает цепную реакцию инстанцирования:

$$F\langle 10\rangle \to F\langle 9\rangle \to \cdots \to F\langle 1\rangle \to F\langle 0\rangle$$

На этапе `Factorial<0>` компилятор выбирает специализацию, где `value` жестко задано как 1. После этого цепочка сворачивается обратно, перемножая константы.

> **Важно!**
>
> Глубина рекурсии шаблонов ограничена компилятором (обычно 900-1024 уровня). Превышение этого лимита приводит к ошибке компиляции. Это одна из главных проблем классического TMP.

Ветвление времени компиляции (Static If)

В классическом TMP нельзя написать `if`, так как инструкции процессора не существуют на этапе компиляции. Ветвление реализуется через выбор одного из двух типов.

```cpp
// Базовый шаблон (не определен или false-ветка)
template <bool Condition, typename T, typename F>
struct If {
    using type = F; // По умолчанию выбираем False-тип
};

// Частичная специализация для true
template <typename T, typename F>
struct If<true, T, F> {
    using type = T; // Если Condition == true, выбираем True-тип
};
```

```
12
13 // Пример использования
14 using ResultType = If<sizeof(void*) == 8, long long, int>::type;
```

Здесь `ResultType` станет `long long` на 64-битной системе и `int` на 32-битной. Это решение является предшественником `std::conditional` из стандартной библиотеки.

Constexpr: Возвращение к императивному стилю

С выходом C++11, а затем C++14 и C++20, подход к метапрограммированию кардинально изменился. Ключевое слово `constexpr` сообщает компилятору, что выражение *может* быть вычислено во время компиляции.

Это позволяет писать метапрограммы на обычном C++, используя привычный синтаксис (циклы, переменные, условные операторы), вместо сложной магии шаблонов.

Эволюция Constexpr

1. **C++11**: Функции `constexpr` были крайне ограничены. Они могли содержать только один оператор `return`. Никаких переменных, циклов или `if`.

```
1 // C++11 стиль
2     constexpr int factorial(int n) {
3         return (n == 0) ? 1 : n * factorial(n - 1);
4     }
```

2. **C++14**: Ограничения сняты. Разрешены локальные переменные, циклы `for/while`, условия `if`, при условии, что они не модифицируют глобальное состояние и не вызывают non-constexpr код.

```
1 // C++14 стиль
2     constexpr int factorial(int n) {
3         int result = 1;
4         for (int i = 1; i <= n; ++i) {
5             result *= i;
6         }
7         return result;
8     }
```

Теперь тот же самый факториал вычисляется без создания сотен типов `struct`, что значительно снижает нагрузку на компилятор и ускоряет сборку.

Compile-time аллокация памяти (C++20)

До C++20 вся память в `constexpr` контексте должна была быть стековой (локальной). Динамическое выделение памяти (`new/delete`) было запрещено.

C++20 ввел понятие **Transient Allocation** (временная аллокация). Теперь внутри `constexpr` функции можно выделять память через `new`, **при условии**, что эта память будет освобожде-

на (`delete`) до завершения вычисления этой функции. Память "не может вытечь" из этапа компиляции в рантайм.

Это изменение позволило сделать методы `std::vector` и `std::string` `constexpr`. Теперь можно сортировать векторы и конкатенировать строки прямо во время компиляции.

```cpp
#include <vector>
#include <numeric>
#include <algorithm>

constexpr int sum_squares(int n) {
    std::vector<int> v; // Аллокация памяти на этапе компиляции!
    for (int i = 0; i < n; ++i) v.push_back(i);

    // Использование STL алгоритмов
    int sum = 0;
    for (int x : v) sum += x * x;

    return sum; // Вектор уничтожается здесь, память освобождается.
}

// Результат вычисляется компилятором
constexpr int val = sum_squares(10);
```

If Constexpr (C++17)

Одним из важнейших дополнений C++17 стала конструкция `if constexpr`. В отличие от обычного `if`, условие в `if constexpr` должно быть известно на этапе компиляции. Главная особенность: **отбрасываемая ветка не инстанцируется**.

Это решает фундаментальную проблему обобщенного программирования: как написать код, который работает для типов с разными интерфейсами, не вызывая ошибок компиляции.

Проблема обычного if

Рассмотрим функцию, которая возвращает `.size()` контейнера, если метод существует, или 0, если это просто число.

```cpp
template <typename T>
auto get_size_bad(const T& t) {
    if (std::is_integral_v<T>) {
        return 0;
    } else {
        return t.size(); // ОШИБКА КОМПИЛЯЦИИ для int!
    }
}
```

Даже если мы передадим `int`, компилятор обязан скомпилировать обе ветки обычного `if`. При попытке скомпилировать `int.size()` произойдет ошибка, несмотря на то, что эта ветка никогда не выполнится в рантайме.

Решение через if constexpr

Используя `if constexpr`, мы указываем компилятору полностью игнорировать код в неактивной ветке. Он проверяется только на базовый синтаксис, но не на корректность методов типа T.

```cpp
#include <type_traits>

template <typename T>
auto get_size_safe(const T& t) {
    if constexpr (std::is_integral_v<T>) {
        return 0;
    } else {
        // Эта строка компилируется ТОЛЬКО если T не интегральный тип.
        // Для int вызов .size() даже не будет проверяться.
        return t.size();
    }
}
```

Этот механизм позволяет писать компактный обобщенный код, заменяя сложные конструкции SFINAE (которые мы рассмотрим в следующей главе) на читаемые блоки условий.

На заметку

Код внутри `if constexpr` все равно должен быть синтаксически корректным C++. Вы не можете написать там случайный набор символов, даже если ветка отбрасывается.

Резюме раздела

- **Legacy TMP** использует рекурсию шаблонов и специализацию. Это мощный, но трудный для чтения и медленный при компиляции подход.
- **Constexpr** позволяет перенести обычную логику C++ (циклы, переменные) на этап компиляции.
- **C++20** разрешает динамическую память в `constexpr`, делая доступными `std::vector` и `std::string`.
- **if constexpr** позволяет исключать куски кода из компиляции в зависимости от свойств типов, упрощая написание шаблонов.

Глава 3

SFINAE и Концепты: Управление перегрузкой функций

Одной из самых сложных и мощных возможностей шаблонов C++ является управление выбором перегрузок. В обычном программировании перегрузка функций тривиальна: компилятор выбирает функцию, сигнатура которой лучше всего соответствует переданным аргументам.

Однако в шаблонном метапрограммировании возникают ситуации, когда <<лучшее>> с точки зрения компилятора совпадение оказывается семантически некорректным. Нам необходим механизм, позволяющий исключать определенные шаблоны из рассмотрения (из множества перегрузки — **Overload Set**) на основе свойств типов.

Исторически этим механизмом был SFINAE, а в C++20 ему на смену пришли Концепты (Concepts).

Проблема жадной перегрузки

Рассмотрим классическую проблему, с которой сталкиваются разработчики контейнеров, подобных `std::vector`. У вектора есть два похожих конструктора: 1. Конструктор заполнения: создает n элементов со значением val. 2. Конструктор диапазона: копирует элементы из диапазона итераторов $[first, last)$.

```cpp
#include <iostream>
#include <vector>

template <typename T>
class Vector {
public:
    // Конструктор 1: Заполнение (Fill Constructor)
    Vector(size_t n, const T& val) {
        std::cout << "Fill Constructor called" << std::endl;
    }

    // Конструктор 2: Диапазон (Range Constructor)
    template <typename InputIter>
    Vector(InputIter first, InputIter last) {
        std::cout << "Range Constructor called" << std::endl;
```

```
        // Представим, что здесь происходит разыменование
        // *first;
    }
};

int main() {
    // Case A: Работает ожидаемо
    Vector<int> v1(5, 10);
    // Аргументы: (int, int).
    // Конструктор 1 ждет (size_t, int). Требуется конверсия int -> size_t.
    // Конструктор 2 ждет (T, T). T выводится как int. Точное совпадение!

    // Компилятор выбирает Конструктор 2.
    // Внутри он пытается сделать *first (разыменовать число 5).
    // ОШИБКА КОМПИЛЯЦИИ: invalid type argument of unary '*'
}
```

В примере выше мы хотели создать вектор из 5 элементов со значением 10. Однако компилятор C++ следует строгим правилам разрешения перегрузки (Overload Resolution).

Анализ ситуации для `Vector<int> v1(5, 10)`:

- **Конструктор 1:** Ожидает `size_t, const int&`. Мы передаем `int, int`. Требуется стандартное преобразование типа (`int` → `size_t`).
- **Конструктор 2:** Шаблонный. Компилятор выводит тип `InputIter = int`. Сигнатура становится `(int, int)`. Это **точное совпадение** (Exact Match).

Точное совпадение всегда приоритетнее преобразования типов. Компилятор выбирает Конструктор 2. Затем он инстанцирует его тело. В теле мы пытаемся обращаться с `int` как с итератором (разыменовываем его). Это приводит к ошибке компиляции внутри тела функции.

Это **Hard Error**. Компиляция прерывается. Нам же нужно, чтобы компилятор, поняв, что `int` не может быть итератором, просто молча проигнорировал этот шаблон и перешел к следующему (Конструктору 1).

SFINAE: Substitution Failure Is Not An Error

Аббревиатура SFINAE (произносится как "сфинэ") расшифровывается как *"Ошибка подстановки не является ошибкой"*. Это правило, зашитое в ядро компилятора C++.

> **Принцип SFINAE**
>
> Если при подстановке выведенных типов в **объявление** (сигнатуру) шаблона функции или класса получается некорректный код, это не приводит к немедленной ошибке компиляции. Вместо этого, данный шаблон просто удаляется из множества кандидатов на перегрузку (Overload Set).

Критически важный нюанс: ошибка должна произойти именно в **заголовке** функции (типы аргументов, возвращаемое значение, параметры шаблона). Если ошибка происходит в **теле** функции (как в примере выше), SFINAE не работает, и мы получаем ошибку компиляции.

Инструмент std::enable_if

Чтобы воспользоваться SFINAE, нам нужно искусственно создать ошибку в заголовке функции, если условие не выполняется. Для этого используется метафункция `std::enable_if`.

```
// Упрощенная реализация enable_if
template <bool B, typename T = void>
struct enable_if {};

// Специализация для true
template <typename T>
struct enable_if<true, T> {
    using type = T;
};
```

Как это работает:

- Если `B == true`, структура имеет вложенный тип `type` (равный `T`).
- Если `B == false`, структура **пуста**. Попытка обратиться к `enable_if<false>::type` вызовет ошибку подстановки.

Применение SFINAE к конструктору

Мы хотим, чтобы шаблонный конструктор существовал только тогда, когда `InputIter` **не** является целочисленным типом.

Стандартный паттерн внедрения `enable_if` в конструкторы (у которых нет возвращаемого значения) — использование дефолтного шаблонного параметра.

```
#include <type_traits>

template <typename T>
class Vector {
public:
    Vector(size_t n, const T& val) { /* ... */ }

    // Этот конструктор будет рассматриваться только если
    // InputIter НЕ является интегральным типом.
    template <typename InputIter,
              typename = std::enable_if_t<!std::is_integral_v<InputIter>>>
    Vector(InputIter first, InputIter last) {
        std::cout << "Range Constructor" << std::endl;
    }
};
```

Разбор механики: 1. Вызов `Vector(5, 10)`. 2. Компилятор пытается инстанцировать шаблонный конструктор. 3. `InputIter` выводится как `int`. 4. Проверяется условие: `!std::is_integral_` Это `false`. 5. Вычисляется `std::enable_if_t<false>`. Такого типа `type` не существует. 6. Происходит ошибка подстановки в параметрах шаблона. 7. Благодаря SFINAE, компилятор **молча отбрасывает** этот конструктор. 8. Единственным оставшимся кандидатом является `Vector(size_t, const T&)`. 9. Аргументы конвертируются, код компилируется корректно.

Detection Idiom и void_t

Часто требуется проверить не просто тип (является ли он числом), а наличие определенного метода или вложенного типа. Например, "есть ли у типа метод `.size()`?".

Для этого используется идиома обнаружения (Detection Idiom) на базе `std::void_t`. `std::void_t<A` — это метафункция, которая всегда превращается в `void`, но требует, чтобы все аргументы `Args...` были валидными типами.

```cpp
// Базовый шаблон: по умолчанию false
template <typename T, typename = void>
struct has_size : std::false_type {};

// Специализация: срабатывает, только если выражение валидно
template <typename T>
struct has_size<T, std::void_t<decltype(std::declval<T>().size())>>
    : std::true_type {};
```

Если у типа `T` есть метод `size()`, специализация становится валидной (второй аргумент `void`). Поскольку специализация более специфична, чем базовый шаблон, компилятор выбирает её, и мы получаем `true_type`. Если метода нет — SFINAE отбрасывает специализацию, и мы падаем в базовый шаблон (`false_type`).

Концепты (Concepts) в C++20

SFINAE — это мощный, но крайне неудобный инструмент. 1. Синтаксис ужасен (угловые скобки, `typename`, `::type`). 2. Сообщения об ошибках нечитаемы. Если ни одна перегрузка не подошла, компилятор вываливает простыню текста о том, почему не сработал `enable_if`. 3. Это "хак" системы типов, а не штатная возможность.

C++20 ввел **Концепты** — прямой способ ограничения шаблонов.

Синтаксис Requires

Вместо `std::enable_if` мы используем ключевое слово `requires` после списка параметров шаблона. Перепишем наш пример с вектором:

```cpp
#include <concepts>

template <typename T>
class Vector {
public:
    Vector(size_t n, const T& val) { /* ... */ }

    // Читаемо и декларативно:
    template <typename InputIter>
    requires (!std::integral<InputIter>) // Ограничение
    Vector(InputIter first, InputIter last) {
        /* ... */
    }
};
```

Более того, стандартная библиотека предоставляет готовый концепт `std::input_iterator`, который проверяет, что тип ведет себя как итератор (поддерживает ++, *, !=).

```
template <std::input_iterator InputIter> // Сокращенный синтаксис
    Vector(InputIter first, InputIter last) { /* ... */ }
```

Если мы теперь попытаемся передать числа в этот конструктор, сообщение об ошибке будет гласить: *"template constraint not satisfied: int is not an input_iterator"*. Это на порядок понятнее, чем ошибки подстановки SFINAE.

Ad-hoc ограничения

Концепты позволяют проверять валидность произвольных выражений прямо по месту, не создавая отдельных структур (как в случае с `void_t`).

```
template <typename T>
requires requires (T x) {
    x.size();          // Должен быть метод size()
    typename T::value_type; // Должен быть вложенный тип value_type
    { x + x } -> std::convertible_to<int>; // Результат сложения приводим к int
}
void process(T obj) {
    // ...
}
```

Выражение `requires (T x) { ... }` создает область видимости, где мы описываем "пробный код". Компилятор проверяет, валиден ли этот код для типа `T`. Этот код никогда не исполняется, только проверяется.

Резюме раздела

- **Overload Resolution** всегда предпочитает точное совпадение типов конверсии. Это опасно для обобщенных конструкторов.
- **SFINAE** позволяет убрать функцию из списка перегрузок, если подстановка типов создает ошибку в сигнатуре.
- **std::enable_if** — основной инструмент SFINAE до C++20. Обычно применяется как дефолтный аргумент шаблона.
- **Concepts (C++20)** полностью заменяют SFINAE. Они делают код чище, а ошибки компиляции — понятными. Всегда предпочитайте `requires` использованию `enable_if` в современном коде.

Глава 4

Архитектура сборки и идиома Pimpl

C++ имеет архаичную модель компиляции, унаследованную от языка C. Программа состоит из набора единиц трансляции (Translation Units), которые компилируются независимо друг от друга, а затем связываются линковщиком. Заголовочные файлы (`.h`) работают через примитивную текстовую подстановку (`#include`).

Эта архитектура порождает проблему физических зависимостей. Любое изменение в заголовочном файле (даже в `private` секции класса) изменяет бинарный интерфейс (ABI) и контрольную сумму файла, вынуждая систему сборки перекомпилировать **все** `.cpp` файлы, которые (прямо или транзитивно) включают этот заголовок. В крупных проектах добавление одного поля `private int x;` в популярный хедер может вызвать "лавину пересборки" (Include Avalanche), занимающую часы.

Идиома Pimpl (Pointer to Implementation) — это архитектурный паттерн, разрывающий эту зависимость путем выноса деталей реализации в отдельный класс, скрытый внутри единицы трансляции.

Анатомия Pimpl

Суть идиомы заключается в замене всех приватных полей класса на единственный непрозрачный указатель (Opaque Pointer) на структуру реализации.

Рассмотрим класс `NetworkClient`, который использует тяжелую библиотеку `<asio.hpp>`.

Без Pimpl (NetworkClient.h):

```cpp
// Проблема: ВЕСЬ asio.hpp попадает к каждому пользователю NetworkClient
#include <asio.hpp>

class NetworkClient {
public:
    void Connect(std::string_view url);
private:
    // Детали реализации "протекают" в интерфейс
    asio::io_context context;
    asio::ip::tcp::socket socket;
};
```

Любой файл, включающий `NetworkClient.h`, будет вынужден парсить тысячи строк `asio.hpp`.

С применением Pimpl:

Мы объявляем структуру `Impl`, но не определяем её в хедере (Forward Declaration).

```
// NetworkClient.h
#include <memory>
#include <string_view>

class NetworkClient {
public:
    NetworkClient();
    ~NetworkClient(); // Важно: деструктор должен быть объявлен!

    void Connect(std::string_view url);

private:
    // Предварительное объявление (Forward Declaration)
    class Impl;

    // Указатель на неполный тип (Incomplete Type)
    std::unique_ptr<Impl> pImpl;
};
```

Теперь хедер ничего не знает о `asio`. Зависимости переносятся в `.cpp` файл.

```
// NetworkClient.cpp
#include "NetworkClient.h"
#include <asio.hpp> // Тяжелый хедер инклюдится только здесь

// Полное определение класса реализации
class NetworkClient::Impl {
public:
    void ConnectInternal(std::string_view url) {
        // Логика работы с asio
    }

    asio::io_context context;
    asio::ip::tcp::socket socket{context};
};

// Конструктор: создаем реализацию
NetworkClient::NetworkClient() : pImpl(std::make_unique<Impl>()) {}

// Деструктор: нужен для unique_ptr (см. далее)
NetworkClient::~NetworkClient() = default;

// Проксирование вызовов
void NetworkClient::Connect(std::string_view url) {
    pImpl->ConnectInternal(url);
}
```

Проблема std::unique_ptr и неполных типов

При использовании сырых указателей (`Impl*`) код выше скомпилировался бы без проблем. Однако ручное управление памятью (`new/delete`) в современном C++ недопустимо. Стандартом де-факто является `std::unique_ptr`.

Здесь возникает тонкий момент, связанный с генерацией деструктора.

Если мы не объявим деструктор `~NetworkClient()` явно в хедере, компилятор попытается сгенерировать его автоматически как `inline` метод. Деструктор `std::unique_ptr<Impl>` вызывает `default_delete<Impl>`, который, в свою очередь, делает `delete ptr`.

Для безопасного вызова `delete` компилятор должен видеть **полное определение типа** `Impl`. Если тип неполный (только forward declaration), оператор `delete` может вызвать Undefined Behavior (если у класса есть нетривиальный деструктор), поэтому `static_assert` внутри `default_delete` выдаст ошибку компиляции: *"invalid application of sizeof to incomplete type"*.

> **Важно!**
>
> Если деструктор генерируется в хедере (автоматически или явно через `= default`), тип `Impl` еще не определен, и компиляция падает.

Решение: 1. Объявить деструктор в хедере: `~NetworkClient();` 2. Определить его в `.cpp` файле, **после** того как определен класс `Impl`: `NetworkClient::~NetworkClient() = default;`

В этой точке (в `.cpp`) тип `Impl` уже является полным (Complete Type), и `unique_ptr` может корректно сгенерировать код удаления.

Fast Pimpl: Избавление от аллокации

Классический Pimpl с `unique_ptr` имеет недостаток производительности: каждый объект требует динамической аллокации памяти (heap allocation). Это создает нагрузку на аллокатор и снижает локальность данных (Pointer Chasing).

Техника **Fast Pimpl** позволяет разместить объект реализации прямо внутри основного объекта (на стеке или внутри его layout), сохраняя инкапсуляцию.

Для этого используется буфер сырой памяти (`std::aligned_storage` или массив `std::byte`), размер и выравнивание которого совпадают с скрытым классом `Impl`.

Реализация Fast Pimpl

В хедере мы резервируем место "вслепую". Нам приходится угадывать размер реализации или фиксировать его константой.

```
// Header
#include <new> // для std::launder и placement new
#include <type_traits>

class FastWidget {
public:
    FastWidget();
    ~FastWidget();
```

```
    void DoWork();

private:
    struct Impl; // Только объявление

    // Константы подбираются экспериментально
    static constexpr size_t ImplSize = 64;
    static constexpr size_t ImplAlign = 8;

    // Сырой буфер памяти
    alignas(ImplAlign) std::byte storage[ImplSize];

    // Вспомогательный метод для каста
    Impl* GetImpl() {
        return reinterpret_cast<Impl*>(&storage);
    }
};
```

В файле реализации мы обязаны проверить, что наши догадки о размере верны, и инициализировать объект через *Placement New*.

```
// CPP file
class FastWidget::Impl {
    int data[10];
    // ... поля реализации
};

// Критическая проверка: если Impl вырастет, компиляция упадет
static_assert(sizeof(FastWidget::Impl) <= sizeof(FastWidget::storage),
              "Storage size is too small for Impl");
static_assert(alignof(FastWidget::Impl) <= alignof(FastWidget::storage),
              "Alignment mismatch");

FastWidget::FastWidget() {
    // Placement new: конструируем объект в буфере storage
    new (&storage) Impl();
}

FastWidget::~FastWidget() {
    // Явный вызов деструктора обязателен!
    GetImpl()->~Impl();
}

void FastWidget::DoWork() {
    GetImpl()->DoWorkInternal();
}
```

Trade-offs Fast Pimpl

- **Преимущество:** Нулевая аллокация в куче. Лучшая локальность кэша (данные лежат рядом).
- **Недостаток 1:** Сложность поддержки. При добавлении полей в `Impl` может сработать

`static_assert`, и придется вручную править константы размера в хедере.

- **Недостаток 2:** Работа с сырой памятью опасна. Забытый вызов деструктора приведет к утечке ресурсов (если `Impl` держит дескрипторы).
- **Недостаток 3:** Strict Aliasing. Использование `reinterpret_cast` требует осторожности. В C++17 для легального доступа к объекту, созданному через placement new, желательно использовать `std::launder`, хотя в данном простом случае доступ через указатель на storage работает на практике.

Резюме раздела

- **Pimpl** разрывает зависимость компиляции, скрывая реализацию за указателем. Это ускоряет сборку и обеспечивает стабильность ABI.
- Использование **std::unique_ptr** с Pimpl требует определения деструктора в `.cpp` файле, чтобы тип реализации был полным.
- **Fast Pimpl** использует буфер на стеке (`alignas`) вместо кучи, убирая оверхед аллокации, но требует ручного управления жизненным циклом (placement new, явный вызов деструктора) и контроля размеров (`static_assert`).

Глава 5

Динамический Полиморфизм: Vtables и RTTI

Полиморфизм — это способность объектов разных типов реагировать на один и тот же вызов метода специфическим для каждого типа образом. В C++ существуют два вида полиморфизма: статический (шаблоны, перегрузка), разрешаемый на этапе компиляции, и динамический (виртуальные функции), разрешаемый во время выполнения (runtime).

Динамический полиморфизм обеспечивает гибкость архитектуры, позволяя работать с объектами через указатель на базовый класс, не зная их реального типа. За эту гибкость приходится платить накладными расходами на вызов (runtime overhead) и память. В этой главе мы разберем низкоуровневую реализацию этого механизма.

Таблица виртуальных методов (vtable)

Ключевое слово `virtual` сообщает компилятору, что связывание вызова функции с ее реализацией должно происходить динамически. Для реализации этого механизма компиляторы (GCC, Clang, MSVC) используют структуру данных, называемую таблицей виртуальных методов (**vtable**).

Модификация раскладки объекта

Как только в классе появляется хотя бы одна виртуальная функция, компилятор неявно добавляет в него скрытое поле — указатель на таблицу виртуальных методов (**vptr**). Обычно `vptr` располагается в самом начале объекта (по смещению 0), чтобы механизм вызова был максимально быстрым.

Рассмотрим иерархию:

```
class Base {
public:
    virtual void func1() { /* Base::func1 */ }
    virtual void func2() { /* Base::func2 */ }
    int x;
};

class Derived : public Base {
public:
```

```
    // Переопределяем func1
    void func1() override { /* Derived::func1 */ }
    // func2 наследуется от Base
    // Добавляем новую виртуальную функцию
    virtual void func3() { /* Derived::func3 */ }
    int y;
};
```

Memory Layout для Base:

- `vptr` → указывает на `Base::vtable`
- `int x`

Содержимое Base::vtable:

- index 0: адрес `Base::func1`
- index 1: адрес `Base::func2`

Memory Layout для Derived:

- `vptr` → указывает на `Derived::vtable`
- `int x` (унаследовано)
- `int y` (собственное)

Содержимое Derived::vtable:

- index 0: адрес `Derived::func1` (переопределена)
- index 1: адрес `Base::func2` (наследуется оригинальная)
- index 2: адрес `Derived::func3` (новая)

Алгоритм диспетчеризации вызова

Когда компилятор встречает вызов `ptr→func1()`, где `ptr` имеет тип `Base*`, он генерирует следующий псевдокод на ассемблере:

1. **Загрузка vptr:** Прочитать значение по адресу, хранящемуся в `ptr` (получаем адрес начала vtable).
2. **Вычисление смещения:** Добавить к адресу vtable смещение, соответствующее индексу метода `func1` (в нашем случае index 0).
3. **Получение адреса функции:** Прочитать адрес из ячейки таблицы.
4. **Вызов (Indirect Call):** Перейти по полученному адресу, передав `ptr` в качестве аргумента `this`.

На заметку

Инициализация `vptr` происходит в конструкторе. Сначала вызывается конструктор `Base`, который устанавливает `vptr` на `Base::vtable`. Затем выполняется тело конструктора `Base`. После этого управление передается конструктору `Derived`, который перезаписывает `vptr` адресом `Derived::vtable`. Именно поэтому вызов виртуальной функции из конструктора никогда не является полиморфным — он вызывает версию текущего конструируемого класса.

Множественное наследование и Pointer Adjustment

Ситуация значительно усложняется при множественном наследовании. Если класс `Derived` наследуется от `Base1` и `Base2`, объект `Derived` должен содержать части обоих родителей, и каждая часть ожидает, что указатель `this` будет указывать на её начало.

```
struct Base1 {
    virtual void method1() {}
    int b1;
};

struct Base2 {
    virtual void method2() {}
    int b2;
};

struct Derived : Base1, Base2 {
    void method1() override {} // override Base1
    void method2() override {} // override Base2
    int d;
};
```

В памяти объект `Derived` будет выглядеть так:

1. **Subobject Base1**:
    - `vptr1` (для Derived-as-Base1)
    - `int b1`
2. **Subobject Base2**:
    - `vptr2` (для Derived-as-Base2)
    - `int b2`
3. **Members Derived**:
    - `int d`

Смещение this (Thunks)

При приведении `Derived*` к `Base2*`, адрес должен измениться. Он должен указывать не на начало всего объекта, а на начало подобъекта `Base2`.

```
Derived* d = new Derived();
Base1* b1 = d; // Адрес совпадает: (void*)b1 == (void*)d
Base2* b2 = d; // Адрес СМЕЩЕН: (void*)b2 == (char*)d + sizeof(Base1)
```

Самое сложное происходит при вызове виртуального метода через `Base2*`. Если мы вызываем `b2→method2()`, который переопределен в `Derived`, функция `Derived::method2` ожидает, что `this` указывает на начало `Derived`, но мы передаем ей указатель на `Base2` (смещенный).

Для решения этой проблемы компилятор генерирует специальную функцию-переходник, называемую **Thunk** (или Trampoline). В vtable для `Base2` лежит не адрес `Derived::method2`, а адрес thunk-а. Этот thunk вычитает смещение из указателя `this` и затем прыгает на реальную функцию `Derived::method2`.

RTTI и dynamic_cast

RTTI (Run-Time Type Information) — это механизм, позволяющий определить реальный тип объекта во время выполнения. Данные RTTI (например, имя типа для `typeid`) обычно хранятся в памяти по указателю, который лежит в специальном слоте vtable (часто по индексу -1).

Оператор dynamic_cast

`dynamic_cast<Target*>(source)` используется для безопасного приведения указателя базового класса к указателю производного класса (Downcast). В отличие от `static_cast`, который просто сдвигает указатель на константу времени компиляции, `dynamic_cast` выполняет проверку в рантайме.

Алгоритм работы: 1. Используя `vptr` объекта `source`, найти RTTI информацию. 2. Проверить, является ли `Target` наследником (или самим типом) реального типа объекта. Это может потребовать обхода дерева наследования. 3. Если приведение возможно, вернуть скорректированный указатель. 4. Если невозможно, вернуть `nullptr` (для указателей) или бросить исключение `std::bad_cast` (для ссылок).

Практический пример: Система событий

Рассмотрим типичный паттерн обработки событий, где полиморфизм используется для стирания типа (Type Erasure) при хранении, а RTTI — для восстановления типа при обработке.

```
#include <string>
#include <iostream>
#include <memory>
#include <vector>

// Интерфейс события.
// Наличие виртуального деструктора обязательно для работы dynamic_cast
// и корректного удаления через базовый указатель.
struct IEvent {
    virtual ~IEvent() = default;
```

```
};

struct ErrorEvent : IEvent {
    std::string message;
};

struct DataEvent : IEvent {
    int payload;
};

// Функция-обработчик не знает конкретных типов на этапе компиляции
void ProcessEvents(const std::vector<std::unique_ptr<IEvent>>& events) {
    for (const auto& event : events) {
        // Попытка интерпретировать событие как ошибку
        if (auto* err = dynamic_cast<ErrorEvent*>(event.get())) {
            std::cout << "[ERROR] " << err->message << std::endl;
            continue;
        }

        // Попытка интерпретировать событие как данные
        if (auto* data = dynamic_cast<DataEvent*>(event.get())) {
            std::cout << "[DATA] Payload: " << data->payload << std::endl;
            continue;
        }

        std::cout << "[INFO] Unknown event type" << std::endl;
    }
}
```

Важно!

`dynamic_cast` работает **только** с полиморфными классами (классами, имеющими хотя бы одну виртуальную функцию). Если в `IEvent` убрать виртуальный деструктор, код не скомпилируется, так как у объекта не будет vtable, а значит, неоткуда взять RTTI.

Чистые виртуальные функции

Если виртуальная функция объявлена с синтаксисом `= 0`, она называется чистой (pure virtual).

```
class Abstract {
public:
    virtual void doWork() = 0;
};
```

Такой класс становится абстрактным: создать его экземпляр невозможно. Однако `vtable` для него все равно может генерироваться (например, для вызова деструктора). В слоты, соответствующие чистым виртуальным функциям, компилятор обычно записывает указатель на служебную функцию-ловушку (например, `__cxa_pure_virtual` в GCC). Если каким-то образом (через UB в конструкторе) вызвать такую функцию, программа аварийно завершится с соответствующим сообщением.

Резюме раздела

- **Vtable** — механизм реализации динамического полиморфизма. Это массив указателей на функции.
- Каждый объект полиморфного класса несет скрытый указатель **vptr**, увеличивающий его размер (обычно на 8 байт).
- Вызов виртуальной функции требует разыменования указателя (индирекции), что может замедлить программу из-за промахов кэша процессора (Branch Prediction Miss).
- При **множественном наследовании** указатель `this` динамически корректируется при приведении типов и вызове методов.
- **dynamic_cast** использует RTTI из vtable для безопасного приведения типов, возвращая `nullptr` при неудаче. Это дорогая операция.

Глава 6

Паттерны проектирования и Идиомы C++

В языке C++ паттерны проектирования выходят за рамки классического объектно-ориентированно программирования (GoF). Благодаря мощной системе шаблонов и детерминированному управлению памятью, многие архитектурные решения реализуются через специфические идиомы (Idioms), недоступные в других языках (Java, C#). В этой главе мы рассмотрим современные реализации порождающих паттернов и статический полиморфизм.

Singleton (Одиночка)

Паттерн Singleton гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к нему. В C++ реализация этого паттерна прошла долгую эволюцию, связанную с проблемами многопоточной инициализации.

Meyers Singleton

До стандарта C++11 безопасная инициализация синглтона в многопоточной среде требовала сложных механизмов блокировок (Double-Checked Locking Pattern), которые часто реализовывались некорректно из-за перестановок инструкций процессором.

Скотт Мейерс предложил элегантное решение, опирающееся на локальные статические переменные.

```
class Database {
public:
    // Удаляем конструкторы копирования и перемещения
    Database(const Database&) = delete;
    Database& operator=(const Database&) = delete;

    // Глобальная точка доступа
    static Database& GetInstance() {
        // "Magic Static"
        static Database instance;
        return instance;
    }

    void Query(const char* sql) { /* ... */ }

private:
```

```
    Database() { /* Тяжелая инициализация подключения */ }
    ~Database() { /* Закрытие соединения */ }
};
```

Механика Magic Statics (Thread-Safe Initialization): Начиная с C++11, стандарт гарантирует: если управление входит в объявление блочной статической переменной (`static` внутри функции), и эта переменная еще не инициализирована, инициализация происходит **потокобезопасно**. Компилятор неявно окружает код инициализации блокировками (обычно через `std::call_once` или атомарные флаги). Если другой поток попытается выполнить `GetInstance()` в момент инициализации, он будет заблокирован до её завершения.

Singleton через CRTP

Чтобы не дублировать код метода `GetInstance` в каждом классе-одиночке, можно использовать CRTP (см. далее) для создания универсального базового класса.

```
template <typename T>
class Singleton {
public:
    static T& GetInstance() {
        static T instance;
        return instance;
    }
    // ... delete copy/move ...
protected:
    Singleton() = default;
    ~Singleton() = default;
};

// Использование:
class Logger : public Singleton<Logger> {
    // friend нужен, чтобы Singleton мог вызвать приватный конструктор Logger
    friend class Singleton<Logger>;
private:
    Logger() { /* ... */ }
};
```

Фабрики в эпоху Smart Pointers

Классические фабрики возвращают сырые указатели. В современном C++ это считается плохой практикой (Ownership Semantics неясна). Стандартом индустрии является возврат `std::unique_ptr`.

```
struct IFruit { virtual ~IFruit() = default; };
struct Apple : IFruit {};
struct Orange : IFruit {};

// Абстрактная фабрика
class FruitFactory {
```

```
public:
    // Возвращаем unique_ptr - передаем владение вызывающему
    static std::unique_ptr<IFruit> Create(std::string_view type) {
        if (type == "apple") return std::make_unique<Apple>();
        if (type == "orange") return std::make_unique<Orange>();
        return nullptr;
    }
};
```

Такой подход гарантирует, что созданный объект будет корректно удален, даже если клиент забудет про него или произойдет исключение.

CRTP: Статический Полиморфизм

Curiously Recurring Template Pattern (CRTP) — идиома, в которой класс `Derived` наследуется от шаблона класса `Base`, параметризованного самим `Derived`.

```
template <typename Derived>
class Base {
    // ...
};

class MyClass : public Base<MyClass> {
    // ...
};
```

Главная цель CRTP — достижение полиморфного поведения без использования виртуальных функций (Static Polymorphism). Базовый класс знает тип наследника на этапе компиляции и может приводить указатель `this` к типу `Derived*`.

```
template <typename Derived>
class BaseAPI {
public:
    void Interface() {
        // Статическое приведение к наследнику.
        // Безопасно, так как мы знаем, что this - это часть объекта Derived.
        static_cast<Derived*>(this)->Implementation();
    }

    // Дефолтная реализация (Compile-time check)
    void Implementation() {
        // Если наследник не переопределил метод, вызовется этот код
    }
};

class Service : public BaseAPI<Service> {
public:
    // "Переопределение" метода (без virtual)
    void Implementation() {
        /* Custom Logic */
    }
```

```
};
```

Преимущества перед `virtual`:

- **Отсутствие vtable:** Экономия памяти (нет vptr) и отсутствие лишней индирекции при вызове.
- **Инлайнинг:** Компилятор видит тело вызываемой функции наследника и может встроить его (Devirtualization), что критично для высоконагруженных циклов.

Пример: Полиморфное клонирование

В классическом ООП для создания копии объекта через базовый указатель требуется виртуальный метод `clone()`. Это приводит к дублированию шаблонного кода в каждом наследнике.

```cpp
struct Shape {
    virtual std::unique_ptr<Shape> clone() const = 0;
    virtual ~Shape() = default;
};

// CRTP Mixin для автоматической реализации clone
template <typename Derived>
struct Cloneable : Shape {
    std::unique_ptr<Shape> clone() const override {
        // Копируем конкретный тип Derived
        return std::make_unique<Derived>(static_cast<const Derived&>(*this));
    }
};

struct Circle : Cloneable<Circle> {
    int radius;
};

struct Square : Cloneable<Square> {
    int side;
};
// Circle и Square автоматически получили корректную реализацию clone()
```

Policy-Based Design

Policy-Based Design (проектирование на основе стратегий) — это подход, популяризированный Андреем Александреску, позволяющий собирать сложные классы из независимых ортогональных поведений (политик) на этапе компиляции.

Вместо того чтобы жестко прописывать поведение внутри класса или использовать паттерн "Стратегия" с виртуальными функциями, мы передаем классы-повеления (Policies) как параметры шаблона.

Реализация умного указателя с политиками

Рассмотрим класс `SmartPtr`, поведение которого (проверка на `nullptr` при доступе) настраивается извне.

```cpp
// Политика 1: Без проверок (для максимальной скорости)
struct NoCheck {
    template <typename T>
    static void Check(T* ptr) {} // Пусто
};

// Политика 2: Строгая проверка
struct EnforceNotNull {
    template <typename T>
    static void Check(T* ptr) {
        if (!ptr) throw std::runtime_error("Null pointer access");
    }
};

// Хост-класс
template <typename T, typename CheckingPolicy>
class SmartPtr : private CheckingPolicy { // Наследование для EBO
    T* ptr = nullptr;
public:
    T* operator->() {
        // Вызов метода политики
        CheckingPolicy::Check(ptr);
        return ptr;
    }
    // ... конструкторы ...
};

// Использование
using FastPtr = SmartPtr<int, NoCheck>;
using SafePtr = SmartPtr<int, EnforceNotNull>;
```

Этот подход позволяет генерировать код, идеально оптимизированный под конкретную задачу. `FastPtr::operator->` скомпилируется в одну инструкцию (чтение адреса), так как пустая функция `NoCheck::Check` будет удалена оптимизатором. `SafePtr` же будет содержать инструкции проверки.

Резюме раздела

- **Singleton** в современном C++ реализуется через локальную статическую переменную (Meyers Singleton), что гарантирует потокобезопасность без мьютексов в пользовательском коде.
- **CRTP** позволяет реализовать статический полиморфизм, заменяя runtime-оверхед виртуальных функций на compile-time разрешение типов.
- **Policy-Based Design** дает возможность создавать гибкие, конфигурируемые компоненты, комбинируя небольшие классы-стратегии через шаблоны.

Глава 7

Case Study: Разработка Интерпретатора Scheme

Разработка интерпретатора функционального языка (диалекта Lisp) является классической задачей, объединяющей все ключевые темы данного курса: полиморфизм, работу с динамической памятью, парсинг и алгоритмы обхода графов.

В этой главе мы спроектируем архитектуру интерпретатора языка Scheme, пройдя путь от токенизации до реализации собственного сборщика мусора (Garbage Collector), необходимого для решения проблемы циклических ссылок, с которой не справляются стандартные умные указатели C++.

Архитектура конвейера (Pipeline)

Процесс интерпретации разделяется на четыре изолированные стадии:

1. **Tokenizer (Lexer):** Преобразует сырой поток символов (`std::istream`) в поток атомарных лексем (токенов). На этом этапе отбрасываются комментарии и пробельные символы, а последовательности цифр группируются в числа.
2. **Parser:** Выполняет синтаксический анализ потока токенов и строит абстрактное синтаксическое дерево (AST). Для Lisp-подобных языков AST совпадает со структурой данных языка (S-expressions).
3. **Evaluation:** Рекурсивный обход AST с вычислением результатов. Здесь реализуется арифметика, вызовы функций и специальные формы (`if`, `define`, `lambda`).
4. **Memory Management:** Сквозной слой, отвечающий за время жизни объектов AST.

Представление данных и AST

В языке Scheme <<код есть данные>> (homoiconicity). Базовым строительным блоком является **Cons Cell** (пара) — структура, содержащая два указателя: `car` (голова) и `cdr` (хвост). Списки строятся как цепочки вложенных пар.

Для представления динамически типизированных объектов Scheme в статически типизированном C++ мы используем полиморфную иерархию классов.

```cpp
#include <memory>
#include <string>
#include <vector>

// Базовый класс для всех объектов Scheme
struct Object {
    virtual ~Object() = default;

    // Поддержка сборщика мусора (см. далее)
    bool marked = false;
};

// Числовой литерал
struct Number : Object {
    int value;
    Number(int v) : value(v) {}
};

// Символ (идентификатор переменной)
struct Symbol : Object {
    std::string name;
    Symbol(const std::string& n) : name(n) {}
};

// Пара (Cons Cell).
// Основной структурный элемент списков и деревьев.
struct Cell : Object {
    std::shared_ptr<Object> first;
    std::shared_ptr<Object> second;

    Cell(std::shared_ptr<Object> f, std::shared_ptr<Object> s)
        : first(f), second(s) {}
};
```

Использование `std::shared_ptr<Object>` кажется естественным выбором. Объекты в Lisp передаются по ссылке, могут использоваться в нескольких местах одновременно (structure sharing), и должны удаляться, когда на них больше никто не ссылается. Механизм подсчета ссылок (Reference Counting) идеально подходит... пока не появляются циклы.

Проблема циклических ссылок

Модель владения `shared_ptr` гарантирует удаление объекта только тогда, когда счетчик ссылок (refcount) достигает нуля. В функциональном программировании возможно создание структур, ссылающихся сами на себя.

Рассмотрим классический пример создания цикла через мутацию хвоста списка (функция `set-cdr!`):

```cpp
// Scheme:
// (define x (list 1 2)) ; x -> (1 . (2 . null))
// (set-cdr! (cdr x) x)  ; хвост списка теперь указывает на начало
```

На уровне C++ это выглядит так:

```cpp
void CreateCycle() {
    auto n1 = std::make_shared<Number>(1);
    auto n2 = std::make_shared<Number>(2);

    // Создаем список (1 2)
    auto cell2 = std::make_shared<Cell>(n2, nullptr);
    auto cell1 = std::make_shared<Cell>(n1, cell2); // cell1 -> cell2

    // Замыкаем цикл: cell2 -> cell1
    // Теперь refcount у обоих объектов равен 2 (1 внешний + 1 внутренний)
    cell2->second = cell1;

    // При выходе из функции внешние указатели (cell1, cell2) уничтожаются.
    // Refcount обоих объектов уменьшается до 1.
    // Память НЕ освобождается. УТЕЧКА.
}
```

В языках без ручного управления памятью (как Scheme) программист не должен думать о разрыве циклов (в отличие от использования `weak_ptr` в C++). Единственным решением является реализация полноценного сборщика мусора.

Реализация Garbage Collector (Mark-and-Sweep)

Алгоритм Mark-and-Sweep (Пометь и Вымети) является классическим подходом к GC. Он не полагается на счетчики ссылок, а определяет <<живучесть>> объектов на основе их достижимости из корневого набора (Roots).

Архитектура меняется: 1. Мы отказываемся от `std::shared_ptr` в полях `Cell`. Используем сырые указатели `Object*`. 2. Создаем класс `Heap` (Куча), который владеет всеми созданными объектами (через `std::vector<std::unique_ptr<Object>>` или свой аллокатор).

Алгоритм

Фаза 1: Mark (Разметка) Обходим граф объектов, начиная с корней (переменные на стеке, глобальное окружение). Помечаем каждый посещенный объект флагом `marked = true`.

```cpp
void Heap::Mark(Object* obj) {
    if (!obj || obj->marked) return;

    obj->marked = true;

    // Если это пара, рекурсивно помечаем детей
    if (auto cell = dynamic_cast<Cell*>(obj)) {
        Mark(cell->first);
        Mark(cell->second);
    }
}
```

Фаза 2: Sweep (Выметание) Проходим по всем объектам, зарегистрированным в куче.

- Если `marked == true`: объект достижим. Сбрасываем флаг в `false` (для следующего цикла GC).

- Если `marked` == `false`: объект недостижим (мусор). Удаляем его.

```
void Heap::Sweep() {
    auto it = allocated_objects.begin();
    while (it != allocated_objects.end()) {
        Object* obj = *it;
        if (obj->marked) {
            obj->marked = false; // Сброс для следующего GC
            ++it;
        } else {
            delete obj; // Освобождение памяти
            // Удаление из списка живых объектов (swap-and-pop для O(1))
            *it = allocated_objects.back();
            allocated_objects.pop_back();
        }
    }
}
```

Этот подход корректно обрабатывает любые циклы, так как недостижимый циклический граф просто не будет помечен в фазе Mark и будет целиком удален в фазе Sweep.

Синтаксический анализ: Recursive Descent

Парсинг Lisp-выражений упрощается благодаря их префиксной скобочной структуре. Метод рекурсивного спуска идеально ложится на грамматику.

Грамматика (упрощенно):

- *Expression* $\rightarrow$ *Atom* | *List*
- *List* $\rightarrow$ '(' *Elements* ')'
- *Elements* $\rightarrow$ *Expression Elements* | ϵ

Реализация парсера:

```
Object* Read(Tokenizer& tokenizer) {
    Token token = tokenizer.Next();

    if (token.type == Token::NUMBER) {
        return new Number(token.int_val);
    }

    if (token.type == Token::OPEN_PAREN) {
        return ReadList(tokenizer);
    }

    if (token.type == Token::SYMBOL) {
        return new Symbol(token.str_val);
    }

    throw SyntaxError("Unexpected token");
}

```

```
Object* ReadList(Tokenizer& tokenizer) {
    Token peek = tokenizer.Peek();
    if (peek.type == Token::CLOSE_PAREN) {
        tokenizer.Next(); // Поглощаем ')'
        return nullptr;   // Пустой список
    }

    Object* head = Read(tokenizer);
    Object* tail = ReadList(tokenizer);

    return new Cell(head, tail);
}
```

Функция `ReadList` рекурсивно вызывает `Read` для считывания головы списка, а затем саму себя для считывания хвоста, автоматически формируя цепочку `Cell`-ов.

Резюме раздела

- Интерпретатор Scheme демонстрирует необходимость использования полиморфизма для реализации динамической типизации в C++.
- **std::shared_ptr** непригоден для графов объектов с циклами без ручного разрыва связей.
- **Mark-and-Sweep** — фундаментальный алгоритм сборки мусора, решающий проблему циклов через анализ достижимости графа.
- Парсинг S-выражений тривиально реализуется методом рекурсивного спуска, отражая рекурсивную природу самого списка.