

# Курс лекций

По языку программирования C++

*Автор:* Владимиров К. И.

Страница с последней редакцией:

[sourceforge.net/projects/cpp-lects-rus](http://sourceforge.net/projects/cpp-lects-rus)

Email автора: [konstantin.vladimirov@gmail.com](mailto:konstantin.vladimirov@gmail.com)

8 августа 2021 г. L<sup>A</sup>T<sub>E</sub>X

# Оглавление

|  |           |
|--|-----------|
| <b>1 Общие сведения</b>  | <b>14</b> |
| 1.1 Стандартизация C++ . . . . .   | 15        |
| 1.1.1 Важность стандартизации . . . . .                                    | 16        |
| 1.1.2 Конформный код . . . . .   | 18        |
| 1.2 Неопределенное поведение . . . . .                                     | 20        |
| 1.2.1 Неявное UB как путь к проклятию . . . . .                            | 20        |
| 1.2.2 Типичные случаи неопределенного поведения . . . . .                  | 21        |
| 1.2.3 Пример исправления ситуации . . . . .                                | 22        |
| 1.2.4 Поиск UB с помощью санитайзеров . . . . .                            | 23        |
| 1.2.5 Статическая верификация кода . . . . .                               | 24        |
| 1.3 Поведение, зависящее от реализации . . . . .                           | 25        |
| 1.4 C++ это конгломерат языковых возможностей . . . . .                    | 26        |
| 1.5 Домашняя наработка по стандартам и неопределенному поведению . . . . . | 29        |
| <b>2 Корни, кровавые корни</b>   | <b>32</b> |
| 2.1 Простые задачи для языка C . . . . .                                   | 32        |
| 2.1.1 Простые конструкции в простых программах . . . . .                   | 32        |
| 2.1.2 Структура программ и заголовочные файлы . . . . .                    | 33        |
| 2.1.3 Дьяволы деталей синтаксиса C . . . . .                               | 35        |
| 2.1.4 Чтение объявлений, как источник радости . . . . .                    | 36        |
| 2.1.5 Ваш друг typedef . . . . .   | 37        |

## ОГЛАВЛЕНИЕ 2

|       |  |    |
|-------|--|----|
| 2.1.6 | Мелкие шероховатости . . . . .   | 38 |
| 2.2   | Борьба с препроцессором . . . . .                                      | 40 |
| 2.2.1 | От макросов к константам времени компиляции . .                        | 40 |
| 2.2.2 | Внешние, статические и встраиваемые функции . .                        | 41 |
| 2.2.3 | Когда нужен препроцессор . . . . .                                     | 44 |
| 2.3   | Базовые концепции языка . . . . .                                      | 47 |
| 2.3.1 | Объявления и определения . . . . .                                     | 47 |
| 2.3.2 | Область видимости и время жизни . . . . .                              | 49 |
| 2.3.3 | Полные и неполные типы . . . . .                                       | 52 |
| 2.3.4 | Lvalue и rvalue . . . . .  | 52 |
| 2.4   | Массивы и указатели . . . . .  | 54 |
| 2.4.1 | Вспоминаем указатели . . . . .   | 54 |
| 2.4.2 | Арифметика указателей . . . . .  | 57 |
| 2.4.3 | Нулевые указатели . . . . .  | 57 |
| 2.4.4 | Действия с массивами . . . . .   | 58 |
| 2.4.5 | Инициализация массивов и указателей . . . . .                          | 60 |
| 2.4.6 | Прогулки за границами . . . . .  | 61 |
| 2.4.7 | Многомерные массивы . . . . .  | 63 |
| 2.5   | От указателей к ссылкам . . . . .                                      | 66 |
| 2.5.1 | Использование ссылок . . . . .   | 66 |
| 2.5.2 | Висячие ссылки . . . . .   | 68 |
| 2.5.3 | Когда ссылки уступают указателям . . . . .                             | 69 |
| 2.6   | Искажение имён и его последствия . . . . .                             | 71 |
| 2.6.1 | Манглирование и перегрузка . . . . .                                   | 71 |
| 2.6.2 | Правила разрешения перегрузки . . . . .                                | 73 |
| 2.6.3 | Аргументы по умолчанию . . . . .                                       | 76 |
| 2.6.4 | Структуры в С и в С++ . . . . .  | 77 |
| 2.7   | От программирования в стиле С к программированию в стиле С++ . . . . . | 79 |

|          |   |            |
|----------|---|------------|
| 2.7.1    | От ввода-вывода через функции к потокам . . . . .   | 79         |
| 2.7.2    | От строк в стиле С к строкам в стиле С++ . . . . .  | 81         |
| 2.7.3    | От макросов к шаблонным функциям . . . . .          | 83         |
| 2.7.4    | От malloc и free к new и delete . . . . .           | 83         |
| 2.7.5    | От приведения в стиле С к приведению в стиле С++    | 85         |
| 2.8      | Ошибки и исключения . . . . .                       | 88         |
| 2.8.1    | Поддержка исключений в языке . . . . .              | 89         |
| 2.8.2    | Отказ от исключений . . . . .                       | 90         |
| 2.8.3    | Исключения под капотом . . . . .                    | 91         |
| 2.9      | Пространства имён . . . . .                         | 93         |
| 2.9.1    | Использование пространств имён и алиасинг . . . . . | 94         |
| 2.9.2    | Поиск Кёнига . . . . .                              | 96         |
| 2.9.3    | Встраиваемые пространства имён . . . . .            | 99         |
| 2.10     | Структуры и объединения . . . . .                   | 101        |
| 2.10.1   | Объединения и каламбуры . . . . .                   | 101        |
| 2.10.2   | Структуры и выравнивание . . . . .                  | 102        |
| 2.10.3   | Прокачка перечислений . . . . .                     | 105        |
| 2.11     | Автоматический вывод типов . . . . .                | 107        |
| 2.11.1   | Auto и Decltype . . . . .                           | 107        |
| 2.11.2   | Расширенный синтаксис функций . . . . .             | 111        |
| 2.11.3   | Обобщения вывода типов функциями . . . . .          | 114        |
| 2.11.4   | Точный вывод и прозрачная оболочка . . . . .        | 116        |
| 2.11.5   | Decaying и минимальные общие типы . . . . .         | 116        |
| 2.12     | Домашняя наработка по базовому С++ . . . . .        | 119        |
| <b>3</b> | <b>Объектно-ориентированное счастье</b>             | <b>122</b> |
| 3.1      | Инкапсуляция и игра в мячик . . . . .               | 123        |
| 3.1.1    | Конкретные классы . . . . .                         | 124        |
| 3.1.2    | POD и NPOD типы . . . . .                           | 125        |

|       |  |     |
|-------|--|-----|
| 3.1.3 | Инициализация и уничтожение . . . . .                  | 126 |
| 3.1.4 | Неявное преобразование типов и explicit . . . . .      | 128 |
| 3.1.5 | Value-инициализация и Default-инициализация . . .      | 129 |
| 3.1.6 | Селекторы . . . . .                                    | 130 |
| 3.1.7 | Статические члены в классе . . . . .                   | 131 |
| 3.1.8 | Указатели на методы класса . . . . .                   | 133 |
| 3.1.9 | Объявления и определения классов . . . . .             | 134 |
| 3.2   | Классы для управления ресурсами . . . . .              | 135 |
| 3.2.1 | Идиома RAII . . . . .                                  | 139 |
| 3.2.2 | Переопределение копирования . . . . .                  | 140 |
| 3.2.3 | Оптимизации возвращаемого значения . . . . .           | 144 |
| 3.3   | Перегрузка операторов . . . . .                        | 146 |
| 3.3.1 | Операторы, формирующие цепочки . . . . .               | 147 |
| 3.3.2 | Симметричные бинарные операторы . . . . .              | 148 |
| 3.3.3 | Инкремент и декремент . . . . .                        | 149 |
| 3.3.4 | Вызов и индексация . . . . .                           | 150 |
| 3.3.5 | Разыменование и его варианты . . . . .                 | 151 |
| 3.3.6 | Приведение типов . . . . .                             | 153 |
| 3.4   | Выделение и освобождение динамической памяти . . . . . | 156 |
| 3.4.1 | Глобальные операторы . . . . .                         | 156 |
| 3.4.2 | Стандартные и специальные формы new . . . . .          | 158 |
| 3.4.3 | Список нормальных форм . . . . .                       | 160 |
| 3.5   | Правые ссылки и семантика перемещения . . . . .        | 161 |
| 3.5.1 | Правые и левые значения . . . . .                      | 161 |
| 3.5.2 | Ссылочное связывание . . . . .                         | 163 |
| 3.5.3 | Провисание ссылок . . . . .                            | 166 |
| 3.5.4 | Семантика перемещения . . . . .                        | 168 |
| 3.5.5 | Свёртка ссылок . . . . .                               | 173 |
| 3.5.6 | Пробрасывание, близкое к совершенству . . . . .        | 175 |

|       |   |     |
|-------|---|-----|
| 3.5.7 | Категории значений . . . . .                                    | 180 |
| 3.6   | Наследование интерфейса . . . . .                               | 183 |
| 3.6.1 | Чисто виртуальные функции и абстрактные классы                  | 184 |
| 3.6.2 | Статический и динамический тип . . . . .                        | 188 |
| 3.6.3 | Подробнее о чисто виртуальных функциях . . . . .                | 189 |
| 3.6.4 | Параметры по умолчанию . . . . .                                | 192 |
| 3.6.5 | Виртуальные деструкторы . . . . .                               | 193 |
| 3.6.6 | Тела для чисто виртуальных функций . . . . .                    | 194 |
| 3.6.7 | Виртуальные конструкторы . . . . .                              | 195 |
| 3.7   | Наследование реализации . . . . .                               | 197 |
| 3.7.1 | Проблема срезки . . . . .                                       | 199 |
| 3.7.2 | Идиома NVI и разделение обязанностей . . . . .                  | 201 |
| 3.7.3 | Тонкости скрытия имён . . . . .                                 | 203 |
| 3.7.4 | Закрытое и защищенное наследование . . . . .                    | 204 |
| 3.7.5 | Конструкторы базовых классов . . . . .                          | 205 |
| 3.7.6 | Больше возможностей для полиморфизма . . . . .                  | 206 |
| 3.8   | Множественное наследование . . . . .                            | 209 |
| 3.8.1 | Виртуальные функции при множественном наследовании . . . . .    | 209 |
| 3.8.2 | Ромбовидные схемы и виртуальные базовые классы                  | 211 |
| 3.8.3 | Динамическое приведение и RTTI . . . . .                        | 214 |
| 3.8.4 | Информация о типах и идентификаторы типов . . . . .             | 216 |
| 3.8.5 | Сюрпризы в конструкторах при виртуальном наследовании . . . . . | 217 |
| 3.8.6 | Порядок инициализации в сложных диаграммах . . . . .            | 218 |
| 3.8.7 | Делегация сестринскому типу . . . . .                           | 219 |
| 3.8.8 | Вложенные классы и снова о пространствах имён . . . . .         | 220 |
| 3.9   | Скажи мне кто твой друг . . . . .                               | 222 |
| 3.9.1 | Обычная дружба . . . . .  | 222 |

|  |            |
|--|------------|
| 3.9.2 Адвокат дружбе не помеха . . . . .   | 224        |
| 3.9.3 Дружба в иерархиях классов . . . . .                                       | 224        |
| 3.9.4 Дружба и виртуальные вызовы . . . . .                                      | 225        |
| 3.10 Основные принципы ООП . . . . .   | 227        |
| 3.10.1 Принцип единственной обязанности . . . . .                                | 227        |
| 3.10.2 Принцип открытости и закрытости . . . . .                                 | 228        |
| 3.10.3 Принцип подстановки Лисков . . . . .                                      | 229        |
| 3.10.4 Принцип разделения интерфейса . . . . .                                   | 230        |
| 3.10.5 Принцип инверсии зависимостей . . . . .                                   | 231        |
| 3.10.6 Закон Деметры . . . . .   | 231        |
| 3.10.7 Проблемы, возникающие при проектировании открытого наследования . . . . . | 232        |
| 3.11 Паттерны объектно-ориентированного программирования .                       | 233        |
| 3.11.1 Порождающие паттерны . . . . .  | 233        |
| 3.11.2 Структурные паттерны . . . . .  | 234        |
| 3.11.3 Паттерны поведения . . . . .  | 235        |
| 3.12 Домашняя наработка по ООП . . . . .   | 238        |
| <b>4 Особая шаблонная магия</b>  | <b>242</b> |
| 4.1 Шаблоны функций . . . . .  | 243        |
| 4.1.1 Вывод типов шаблонами . . . . .  | 244        |
| 4.1.2 Обобщенность и неожиданности . . . . .                                     | 245        |
| 4.1.3 Перегрузка шаблонных функций . . . . .                                     | 247        |
| 4.1.4 Управление инстанцированием шаблонов функций .                             | 249        |
| 4.1.5 Два полиморфизма . . . . .   | 252        |
| 4.1.6 Когда C++ быстрее, чем C . . . . .   | 254        |
| 4.1.7 Когда шаблонный полиморфизм уступает динамическому . . . . .               | 255        |
| 4.2 Шаблоны классов и специализация . . . . .                                    | 257        |
| 4.2.1 Специализация . . . . .  | 258        |

|       |  |     |
|-------|--|-----|
| 4.2.2 | Частичная специализация . . . . .                | 260 |
| 4.2.3 | Шаблонные параметры по умолчанию . . . . .       | 262 |
| 4.2.4 | Вывод типов против подстановки типов . . . . .   | 262 |
| 4.2.5 | Разрешение имён . . . . .                        | 263 |
| 4.2.6 | Устранение неоднозначности . . . . .             | 266 |
| 4.2.7 | Целочисленные шаблонные параметры . . . . .      | 268 |
| 4.2.8 | Параметризация шаблонов указателями и ссылками   | 269 |
| 4.2.9 | Шаблоны псевдонимов . . . . .                    | 272 |
| 4.3   | Шаблоны и ООП . . . . .                          | 274 |
| 4.3.1 | Специализация методов класса . . . . .           | 274 |
| 4.3.2 | Переходники типов . . . . .                      | 275 |
| 4.3.3 | Шаблоны членов и инкапсуляция . . . . .          | 277 |
| 4.3.4 | Шаблоны полей и наследование . . . . .           | 278 |
| 4.3.5 | Параметризация виртуальности . . . . .           | 279 |
| 4.3.6 | Шаблоны и дружба . . . . .                       | 279 |
| 4.4   | CRTP . . . . .                                   | 282 |
| 4.4.1 | Основная идея . . . . .                          | 282 |
| 4.4.2 | CRTP и развязывание . . . . .                    | 285 |
| 4.4.3 | Шаблонные шаблонные параметры . . . . .          | 286 |
| 4.4.4 | CRTP с закрытой базой и ещё немного дружбы . . . | 288 |
| 4.5   | Вариабельные шаблоны . . . . .                   | 290 |
| 4.5.1 | Поговорим о «...» . . . . .                      | 290 |
| 4.5.2 | Пачки параметров . . . . .                       | 291 |
| 4.5.3 | Паттерны раскрытия пачек параметров . . . . .    | 292 |
| 4.5.4 | Emplace и доброе волшебство . . . . .            | 295 |
| 4.5.5 | Раскрытие с помощью списка инициализации . . .   | 297 |
| 4.5.6 | Наконец-то безопасный printf . . . . .           | 299 |
| 4.5.7 | Наконец-то совершенно прозрачная оболочка . . .  | 300 |
| 4.6   | Кортежи . . . . .                                | 301 |

|        |   |     |
|--------|---|-----|
| 4.6.1  | Слияние ежа и ужа . . . . .                                       | 301 |
| 4.6.2  | Стандартные кортежи и их применения . . . . .                     | 302 |
| 4.6.3  | Отложенное исполнение на кортежах . . . . .                       | 304 |
| 4.6.4  | Кортежи для обработки вариабельных шаблонных параметров . . . . . | 305 |
| 4.7    | Lambda expressions . . . . .                                      | 307 |
| 4.7.1  | Generic lambdas . . . . .   | 308 |
| 4.7.2  | Замыкания . . . . .   | 309 |
| 4.7.3  | Захват контекста в подробностях . . . . .                         | 311 |
| 4.7.4  | Перегруженные лямбда-выражения . . . . .                          | 314 |
| 4.7.5  | Типизация лямбда выражений: function type . . . . .               | 316 |
| 4.7.6  | Захват как способ уменьшения связности . . . . .                  | 318 |
| 4.7.7  | Провисание ссылок на захваченный контекст . . . . .               | 321 |
| 4.7.8  | Связывание аргументов . . . . .                                   | 322 |
| 4.7.9  | Variadic lambdas и монады . . . . .                               | 324 |
| 4.8    | Правила инстанцирования и SFINAE . . . . .                        | 329 |
| 4.8.1  | Инстанцирование шаблонов . . . . .                                | 329 |
| 4.8.2  | Достоинства ленивости . . . . .                                   | 332 |
| 4.8.3  | Идиома SFINAE . . . . .   | 334 |
| 4.9    | Систематическое SFINAE . . . . .                                  | 339 |
| 4.9.1  | Истина и ложь . . . . .   | 339 |
| 4.9.2  | Определители и модификаторы . . . . .                             | 340 |
| 4.9.3  | Вариабельные шаблоны для списков типов . . . . .                  | 343 |
| 4.9.4  | Статический ассерт . . . . .                                      | 345 |
| 4.9.5  | Условный тип и условный Enable . . . . .                          | 347 |
| 4.10   | Метапрограммирование . . . . .                                    | 350 |
| 4.10.1 | Простая рекурсия и арифметика . . . . .                           | 350 |
| 4.10.2 | Две модели вычислений . . . . .                                   | 351 |
| 4.10.3 | Ветвления и пример квадратного корня . . . . .                    | 355 |

|          |   |            |
|----------|---|------------|
| 4.10.4   | Простые числа . . . . .                                 | 356        |
| 4.11     | Вычисления времени компиляции . . . . .                 | 358        |
| 4.11.1   | Ещё раз о константности . . . . .                       | 358        |
| 4.11.2   | Константно-выраженные функции . . . . .                 | 361        |
| 4.11.3   | Аннотация данных . . . . .                              | 362        |
| 4.11.4   | Темные чудеса шаблонных переменных . . . . .            | 363        |
| 4.11.5   | Аннотация конструкторов . . . . .                       | 364        |
| 4.11.6   | Пользовательские литералы . . . . .                     | 365        |
| 4.12     | Домашняя наработка по шаблонам . . . . .                | 367        |
| <b>5</b> | <b>Лабиринты стандартной библиотеки</b>                 | <b>373</b> |
| 5.1      | Строки . . . . .  | 376        |
| 5.1.1    | Проблемы безопасности в С-строках . . . . .             | 377        |
| 5.1.2    | Принципиальное устройство <code>string</code> . . . . . | 378        |
| 5.1.3    | Проблемы владения содержимым . . . . .                  | 381        |
| 5.1.4    | Строки с совместным использованием . . . . .            | 384        |
| 5.1.5    | Оптимизации для небольших строк . . . . .               | 386        |
| 5.1.6    | Обобщения строк . . . . .                               | 387        |
| 5.2      | Исключения . . . . .                                    | 390        |
| 5.2.1    | Обработка ошибок в конструкторах . . . . .              | 390        |
| 5.2.2    | Размотка стека . . . . .                                | 392        |
| 5.2.3    | Стандартные классы исключений . . . . .                 | 395        |
| 5.2.4    | Нейтральность . . . . .                                 | 398        |
| 5.2.5    | Гарантии безопасности . . . . .                         | 400        |
| 5.2.6    | Влияние исключений на проектирование . . . . .          | 403        |
| 5.2.7    | Двухуровневый контейнер . . . . .                       | 405        |
| 5.2.8    | Жизнь без исключений . . . . .                          | 408        |
| 5.3      | Умные и слишком умные указатели . . . . .               | 412        |
| 5.3.1    | Первая попытка – не слишком умный указатель . .         | 413        |

|       |  |     |
|-------|--|-----|
| 5.3.2 | Автоматические указатели и их проблемы . . . . .   | 415 |
| 5.3.3 | Уникальное владение . . . . .                      | 417 |
| 5.3.4 | Проектирование с уникальными указателями . . . . . | 420 |
| 5.3.5 | Совместное владение . . . . .                      | 422 |
| 5.3.6 | Слабые указатели . . . . .                         | 427 |
| 5.4   | Последовательные контейнеры . . . . .              | 431 |
| 5.4.1 | Возможности контейнеров и требования к элементам   | 432 |
| 5.4.2 | От встроенных массивов к векторам . . . . .        | 433 |
| 5.4.3 | От векторов обратно к массивам . . . . .           | 438 |
| 5.4.4 | Списки инициализации . . . . .                     | 440 |
| 5.4.5 | Первое представление об итераторах . . . . .       | 443 |
| 5.4.6 | Деки . . . . .                                     | 445 |
| 5.4.7 | Списки . . . . .                                   | 447 |
| 5.4.8 | Адаптеры . . . . .                                 | 449 |
| 5.4.9 | Контейнеро-подобные классы . . . . .               | 451 |
| 5.5   | Ввод и вывод . . . . .                             | 454 |
| 5.5.1 | Проблемы с C-style IO . . . . .                    | 455 |
| 5.5.2 | Иерархия классов для IO . . . . .                  | 456 |
| 5.5.3 | Форматирование . . . . .                           | 456 |
| 5.5.4 | Неформатный ввод и вывод . . . . .                 | 459 |
| 5.5.5 | Состояния и обработка ошибок . . . . .             | 460 |
| 5.5.6 | Работа с файлами . . . . .                         | 462 |
| 5.5.7 | Бинарный ввод и вывод . . . . .                    | 465 |
| 5.5.8 | Работа со строками . . . . .                       | 465 |
| 5.5.9 | Буферизация . . . . .                              | 466 |
| 5.6   | Локализация . . . . .                              | 468 |
| 5.6.1 | Символы и характеристики символов . . . . .        | 469 |
| 5.6.2 | Локали . . . . .                                   | 470 |
| 5.6.3 | Фасеты . . . . .                                   | 471 |

|        |  |     |
|--------|--|-----|
| 5.7    | Итераторы . . . . .                              | 472 |
| 5.7.1  | От указателей к итераторам . . . . .             | 472 |
| 5.7.2  | Характеристики итераторов . . . . .              | 477 |
| 5.7.3  | Итераторы потоков ввода-вывода . . . . .         | 480 |
| 5.7.4  | Константные и обратные итераторы . . . . .       | 482 |
| 5.7.5  | Адаптеры итераторов . . . . .                    | 485 |
| 5.7.6  | Валидность и инвалидация итераторов . . . . .    | 486 |
| 5.8    | Алгоритмы . . . . .                              | 490 |
| 5.8.1  | Абстракция циклов . . . . .                      | 490 |
| 5.8.2  | Копирование и суффиксы для алгоритмов . . . . .  | 491 |
| 5.8.3  | Общий обзор . . . . .                            | 493 |
| 5.8.4  | Идиома <code>erase-remove</code> . . . . .       | 496 |
| 5.8.5  | Абстракция циклов . . . . .                      | 498 |
| 5.8.6  | Трансформации . . . . .                          | 498 |
| 5.8.7  | Бинарный поиск и его варианты . . . . .          | 499 |
| 5.8.8  | Алгоритмический подход к перестановкам . . . . . | 500 |
| 5.9    | Ассоциативные контейнеры . . . . .               | 504 |
| 5.9.1  | Множества . . . . .                              | 507 |
| 5.9.2  | Модификация множеств . . . . .                   | 509 |
| 5.9.3  | Отображения . . . . .                            | 511 |
| 5.9.4  | Модификация отображений . . . . .                | 513 |
| 5.9.5  | Словари . . . . .                                | 515 |
| 5.9.6  | Переход к неупорядоченности . . . . .            | 517 |
| 5.10   | Память своими руками . . . . .                   | 520 |
| 5.10.1 | Простые аллокаторы . . . . .                     | 521 |
| 5.10.2 | Характеристики и состояние аллокаторов . . . . . | 525 |
| 5.10.3 | Пример: кэшированное освобождение . . . . .      | 527 |
| 5.10.4 | Локальные аллокаторы и их аренды . . . . .       | 529 |
| 5.10.5 | Проблемы области распространения . . . . .       | 531 |

*ОГЛАВЛЕНИЕ* 12

|  |            |
|--|------------|
| 5.10.6 Полиморфные аллокаторы . . . . .                            | 535        |
| 5.10.7 Иерархия ресурсов в памяти . . . . .                        | 538        |
| 5.10.8 Собственные контейнеры, пользующиеся аллокаторами . . . . . | 539        |
| 5.10.9 Сводная таблица . . . . .                                   | 541        |
| 5.11 Концепты . . . . .  | 543        |
| 5.11.1 Добрые старые способы контроля полиморфизма .               | 545        |
| 5.11.2 Простые ограничения . . . . .                               | 548        |
| 5.11.3 Пример – перегрузка конструкторов . . . . .                 | 550        |
| 5.11.4 Сложные ограничения . . . . .                               | 552        |
| 5.11.5 Простые концепты . . . . .                                  | 555        |
| 5.11.6 Вариабельные концепты . . . . .                             | 560        |
| 5.11.7 Частичное упорядочение . . . . .                            | 562        |
| 5.11.8 Критика концептов . . . . .                                 | 563        |
| 5.11.9 Сны о чем-то большем . . . . .                              | 564        |
| 5.12 Домашняя наработка по стандартной библиотеке . . . . .        | 567        |
| <b>6 Толкаясь невидимыми локтями</b>                               | <b>576</b> |
| 6.1 Нити исполнения и вышивание . . . . .                          | 577        |
| 6.1.1 Параллельное выполнение . . . . .                            | 578        |
| 6.1.2 Обещания и асинхронность . . . . .                           | 581        |
| 6.1.3 Чуть больше про обещания . . . . .                           | 586        |
| 6.2 Конкуренция за данные . . . . .                                | 588        |
| 6.2.1 Гонки . . . . .  | 588        |
| 6.2.2 Простая синхронизация . . . . .                              | 590        |
| 6.2.3 Проблемы интерфейсов . . . . .                               | 592        |
| 6.2.4 Экзотичные блокировки . . . . .                              | 595        |
| 6.2.5 Взаимные блокировки . . . . .                                | 596        |
| 6.2.6 Одноразовые события . . . . .                                | 600        |
| 6.2.7 Условные переменные . . . . .                                | 603        |

*ОГЛАВЛЕНИЕ*

13

|       |  |     |
|-------|--|-----|
| 6.2.8 | Разделяемые блокировки . . . . .                 | 605 |
| 6.3   | Атомарность . . . . .                            | 607 |
| 6.3.1 | Базовая атомарность . . . . .                    | 608 |
| 6.3.2 | Взаимные блокировки . . . . .                    | 610 |
| 6.3.3 | Модели памяти и теория относительности . . . . . | 612 |
| 6.3.4 | Хрупкость атомарности . . . . .                  | 618 |
|       | Список иллюстраций . . . . .                     |     |
|       | Список литературы . . . . .                      |     |

**Предметный указатель**

# Глава 1

## Общие сведения

C++ (произносится “си плас плас”) это язык общего назначения с сильной статической типизацией и ручным управлением ресурсами. Де-факто в современном программировании C++ является индустриальным стандартом. Язык был создан Бъярном Строструпом в 1980-м году как объектно-ориентированное расширение языка C, известного с 70-х. В 1985-м появилась первая коммерческая версия компилятора Cfront и по языку была опубликована первая книга.

По прошествии времени, C++ был принят сообществом и широко распространился. На языке решается большое количество разнообразных задач. Код на C++ переносимо работает на различных архитектурах и операционных системах.

В 1998-м году язык C++ был стандартизован (работа велась с 1990-го года). Сейчас язык C++ имеет развитую инфраструктуру средств компиляции, отладки, библиотек, поддержку в IDE.

Много новых возможностей было добавлено в язык в 2011-м году, с принятием обновлённого стандарта. В частности именно тогда появились правые ссылки, функции времени компиляции, лямбды, был существенно улучшен вывод типов.

Некоторые дополнительные возможности были добавлены в стандартах, принятых в 2014-м и 2017-м годах.

Стандарт, принятый в 2020-м году принёс серьёзные изменения, такие, как модули, сопрограммы и концепты, снова проведя революцию в языке и сделав его ещё более новым и современным.

В настоящее время ведется работа над новым стандартом, принятие

которого запланировано на 2023-й год.

Язык и библиотека в их эволюции оперативно поддерживаются в современных компиляторах, собственно большинство современных промышленных компиляторов просто на C++ и написаны.

## 1.1 Стандартизация C++

*First, I'd like to see the basic tools such as compilers, debuggers, profilers, database interfaces, GUI builders, CAD tools, and so forth fully support the ISO standard*

– Bjarne Stroustrup

**Вопрос к студентам:** что такое ISO, какие стандарты ISO вы знаете (читали).

Международная организация по стандартизации (ISO) создана в 1946 году двадцатью пятью национальными организациями по стандартизации. На сегодняшний день в состав ИСО входит 165 стран. СССР был одним из основателей организации, постоянным членом руководящих органов, дважды представитель Госстандарта избирался председателем организации. Россия стала членом ИСО как правопреемник СССР. С 2005 года Россия входит в Совет ИСО. Официальными языками являются: английский, французский и русский. Конкретные стандарты могут быть опубликованы на одном из этих языков, но не обязательно на всех. Например, стандарт C++ опубликован только на английском языке.

Первый стандарт языка C++ под кодовым номером ISO/IEC 14882-1998 [19] был написан и утверждён ISO в 1998 году. Он включает в себя по нормативной ссылке стандарт языка C, ISO/IEC 9899-1990 [20], что означает следующее:

- Возможности C90 (кроме явно оговоренных исключений) также являются возможностями C++98, также и стандартная библиотека языка C в редакции C90 является стандартной библиотекой C++98
- Новые возможности C99 [20] не являются возможностями C++98 (стоит отметить, что стандарт C99 включен по ссылке в C++11 [24])
- Стандарт C11 [22] снова разошёлся с текущим нижележащим подмножеством C в C++), снова был включен по нормативной ссылке

в C++14 [25]

- В любом стандарте C++ начиная с C++11 есть Annex C, приложение, описывающее, что сломали в сравнении с языком C и с прошлыми версиями языка C++.

Текущий стандарт C++ имеет кодовый номер ISO/IEC 14882-2020

Важно знать какой стандарт по умолчанию поддерживает тот компилятор которым вы пользуетесь (и та его версия, которой вы пользуетесь в настоящий момент). Очень часто по умолчанию компилятор поддерживает не чистый стандарт (особенно в случае языка C) а стандарт с несколькими специфичными для конкретного компилятора расширениями. Скажем GCC 5.1 по умолчанию поддерживает для языка C стандарт C11 + GNU extensions (так называемый gnu11), а для языка C++ стандарт C++98 + GNU extensions (так называемый gnu++98).

Очень часто компиляторы поддерживают даже будущий стандарт. Типичный случай – опция `c++1z` в GCC в 2016-м году включала совместимость со всеми утвержденными на данный момент предложениями в ещё не принятый документ C++17.

**Домашняя наработка:** посмотрите какой стандарт по умолчанию поддерживает ваш компилятор

### 1.1.1 Важность стандартизации

Стандарты и даже ошибки в стандартах отлиты в граните. Например, в стандарте IEEE POSIX 1003.1-1988 была допущена ошибка: в одной главе было указано, что `PATH_MAX` включает terminating null, в то время, как в другой главе было указано что не включает. Ошибка была обнаружена после публикации стандарта, люди обратились в комиссию стандартизации и оттуда пришло объяснение, что если написано и так и так, значит может быть и так и так. Хотя изначально идея была как раз в том, чтобы определённо установить включает ли `PATH_MAX` нулевой символ.

Язык как соглашение между программистами и разработчиками компиляторов утверждается своим стандартом и не существует вне его. Все стандарты ISO доступны бесплатно в версии final draft – последнего черновика перед утверждением. Часто пиратски доступны и окончательные версии, но цена не так велика, а для личного использования разницы с final draft почти нет.

Читать и понимать стандарт языка – то, чем занимается каждый программист при решении любых спорных вопросов, возникающих у него относительно тех или иных языковых средств. Это последняя и окончательная инстанция.

В качестве примера, вы пишете следующий код на чем-то, что кажется вам языком программирования C:

```
1 inline void my_assert(int b, const char *str) {
2     if (b) return;
3     fprintf(stderr, "Assertion failed: %s\n", str);
4     abort();
5 }
6
7 int main (int argc, char **argv) {
8     my_assert (argc > 0, "argc <= 0");
9     return 0;
10 }
```

**Вопрос к студентам:** видите ли вы какие-нибудь проблемы в этом коде?

Объяснение: изменение в стандарте C99 семантики `inline` по сравнению с C++98 делает упомянутую тут `extern inline` функцию необязательной заглушки, которую компилятор имеет право выбросить. Подробности последуют в (2.2.2).

И это только незначительный пример, который не приводит к особенно разрушительным последствиям и причины которого можно найти в интернете сравнительно просто. По незнанию или ориентируясь на “здравый смысл” можно допустить существенно более серьезные просчеты. Все эти лекции можно считать приглашением к самостоятельной проработке стандарта C++ совместно с решением практических задач. Собственно, чтобы научится писать программы, надо знать язык и писать программы, иных вариантов нет.

Очень часто в этом конспекте будут делаться ссылки на пункты стандартов. Чтобы не путать их с главами, внутри круглых скобок будет ссылка на документ стандарта в списке литературы. Поэтому (5.1.2) это ссылка на главу 5.1.2 этого текста, тогда как (C11 5.1.2) это ссылка на пункт 5.1.2 стандарта C11 ([22]).

### 1.1.2 Конформный код

Стандарт языка C++ традиционно отличается от стандарта языка С тем, что стандарт С определяет корректность **программы**, написанной на С, стандарт же C++ определяет корректность **компилятора**. Это различие с первого раза кажется несущественным и бюрократическим, но во многом из-за него, например, стандарт C99 довольно долгое время не был целиком реализован ни в одном компиляторе С (и только недавно стал стандартом по умолчанию для GCC). Конформные программы можно писать и с подмножеством возможностей. В то же время гораздо более сложный C++98 был реализован в некоторых компиляторах даже в своих наиболее эзотерических частях, таких как экспорт шаблонов.

Тем не менее, очень часто бывает проще даже в случае C++ кода говорить именно о качестве кода, поскольку программа это то, что (до определенной степени) можно модифицировать.

Любой код, подаваемый на вход удовлетворяющего стандарту компилятора, принадлежит к одной из следующих категорий:

- **Syntax violation** – нечто, вообще не являющееся кодом на языке C++. Это наиболее распространённый вид кода, производимого программистами.
- **Implementation defined** – зависящий от разработчика и документации конкретного компилятора. Пример – как ведёт себя знаковое целое при сдвиге вправо.
- **Unspecified** – поведение корректного кода, не регламентированное стандартом. Пример – порядок выполнения аргументов у функции.
- **Undefined behavior** – поведение некорректного кода, не запрещённое стандартом. Пример – поведение знакового целого числа при переполнении.
- **Strictly conforming** – отличный код, полностью удовлетворяющий стандарту и не выходящий за пределы стандарта. Редко встречается *in anima vili* и никогда у новичков.
- **Conforming** – код удовлетворяющий стандарту + использующий implementation defined features.

У разных компиляторов есть своя стратегия выдачи диагностических сообщений. Для компиляции большинства примеров из этих лекций, будет использован компилятор g++ версии 9.2.0 или выше, входящий в GNU Compiler Collection. Рекомендуемые опции:

```
g++ -pedantic-errors -Wall -Werror -g3 -O0 --std=c++98  
g++ -pedantic-errors -Wall -Werror -g3 -O0 --std=c++20
```

Эти опции заставляют сообщать о не-conforming коде и воспринимать все диагностические сообщения как ошибки, а также включать отладочную информацию включая отладку по макросам и отключать все оптимизации (с моей точки зрения это единственный приемлемый способ компиляции учебных примеров).

Пример conforming кода на C-подмножестве C++:

```
1 int main() {  
2     (void) printf("biggest int is %d\n", INT_MAX);  
3 }
```

Приведённый выше код является conforming но не strictly conforming из-за использования INT\_MAX, являющегося implementation-defined. Тем не менее, нет способа заставить GCC выдать такое диагностическое предупреждение и этот код проходит компиляцию без ошибок, так что можно считать уровень conforming достаточным.

Обратите внимание на форму функции main. В отличии от C, в C++ считается хорошим тоном не указывать явный void параметром, потому что функция без аргументов в C++ как раз и означает “не принимающая ничего”.

## 1.2 Неопределенное поведение

Из всех рассмотренных выше видов поведения программы, самое страшное это неопределённое поведение (undefined behavior, UB). Обратите внимание: аббревиатура UB здесь и далее употребляется только для undefined behavior, но никогда для unspecified, хотя там сокращение такое же.

Неопределённое поведение **не является злом**: отказ определять поведение в граничных ситуациях даёт компилятору возможность генерировать гораздо более оптимальный код. Иногда программист закладывает логический аналог неопределенного поведения в свою программу сознательно. Например при удалении односвязного списка вы можете сказать, что поведение процедуры удаления не определено если в списке есть петля и именно это ограничение позволит написать удаление эффективно. Аналогично и стандарт каждым своим UB-clause оставляет компилятору свободу для маневра.

Тем не менее, UB является опасностью. Любой программист на языках C и C++ с некоторым опытом развивает чутьё на неопределённые ситуации, но даже профессионалы часто попадают в ловушки. В этих лекциях я предполагаю остановиться на этой теме с самого начала, поскольку упомянутое выше чутьё надо развивать сразу. Сначала эффективный пример.

### 1.2.1 Неявное UB как путь к проклятию

Следующий код на C (взят из SPEC benchmark) выглядит достаточно невинно

```

1 int k, satd = 0, dd, d[16];
2
3 /* ... more code here ... */
4
5 for (dd = d[k = 0]; k < 16; dd = d[++k])
6     satd += (dd < 0 ? -dd : dd);

```

Но в эксперименте с GCC 4.8.0 получается бесконечный цикл. Компилятор на этапе aggressive-loop-optimizations, считает, что без выхода за границы массива k не может быть больше 15. А раз  $15 < 16$ , то и выход по условию  $k < 16$  невозможен, так что его и проверять не стоит.

Этот пример показателен тем, что в случае UB компилятор может не только **сделать** всё что угодно (например отформатировать диск), но и **не сделать** чего-то, причём компилятор даже не обязан об этом сообщать.

### 1.2.2 Типичные случаи неопределенного поведения

Здесь будет рассмотрен только язык С и С-подмножество С++. Проблемы, возникающие при работе с ООП и шаблонами будут рассмотрены в конце соответствующих частей. Типичными случаями неопределенного поведения являются:

- Арифметика

- Целочисленное знаковое переполнение
- Математическая неопределенность результата (скажем целочисленное деление на 0)
- Сдвиг влево на отрицательные значения
- Сдвиг на значения, превышающие размер типа
- Приведение целого числа к не вмещающему его типу
- Попытка изменить константный объект, приведя его к не константному

- Указатели

- Разыменование нулевого указателя
- Разыменование неинициализированного указателя
- Разыменование указателя на память, динамически выделенную с нулевым размером
- Использование указателей и ссылок на объекты с истекшим сроком жизни
- Адресная арифметика с результатом, выходящим за границы массива и доступ к такой памяти
- Преобразование указателей в несовместимые типы
- Использование темсру на пересекающихся участках памяти

- Препроцессор

- Непустой исходный файл, не оканчивающийся на пустую строку (или оканчивающийся на обратный слеш)
- Обратный слеш после которого стоит нечто отличное от перечня escape-кодов
- Выход за реализационные пределы (скажем 1024 условия в switch)
- Динамически генерируемый токен под `#if`

Разумеется этот список не претендует на полноту, но он даёт представление о местах, которых стоит опасаться.

### 1.2.3 Пример исправления ситуации

Очень часто код, демонстрирующий неопределенное поведение, можно переписать на эквивалентный детерминированный. Это работает не во всех случаях, но гораздо чаще, чем многие думают. Типичный грозящий UB код:

```

1 int foo(int a) {
2     assert(a + 100 > a); /* ORLY? */
3     printf("%d %d\n", a + 100, a);
4     return a;
5 }
```

Автор этого кода скорее всего считает, что оградился от случаев арифметического знакового переполнения. На самом деле он просто заложился на неопределенное поведение. Простое переписывание кода сделает его гораздо более безопасным:

```

1 int foo(int a) {
2     assert(a < INT_MAX - 100);
3     printf("%d %d\n", a + 100, a);
4     return a;
5 }
```

Если исправление ситуации с неопределенным поведением в ваших силах – делайте это. Увы, иногда обнаружение неопределенного поведения на этапе компиляции проблематично:

```

1 int foo(int* a, int len) {
2     assert ((a != NULL) && (len > 1));
```

```

3     return a[len/2]; /* ORLY? */
4 }
```

В случае если в `len` передано неверное значение, здесь возможен выход за границы массива, который на этапе компиляции отловить сложно или невозможно. И здесь вступают в игру проверки времени исполнения – так называемые санитайзеры.

#### 1.2.4 Поиск UB с помощью санитайзеров

Некоторые компиляторы (такие как GCC) поддерживают инструментирование кода в местах, где возможно неопределенное поведение. Для GCC доступен вызов с опцией `-fsanitize=undefined`, которая включает UndefinedBehaviorSanitizer – open-source средство определения неопределенного поведения во время исполнения. Он проверяет почти все случаи рассмотренные в (1.2.2), но существенно замедляет программу, так как по сути генерирует вызов проверяющей функции на каждое знаковое суммирование, доступ по указателю, сдвиг и прочее.

Для того, чтобы проверить конкретный случай UB а не все их сразу, можно подать вместо `undefined`, одну из её подопий: `shift`, `integer-divide-by-zero`, `unreachable`, `vla-bound`, `null`, `return`, `signed-integer-overflow`, `bounds`, `bounds-strict`, `alignment` и другие.

Например `fsanitize=integer-divide-by-zero` расставит проверки только на случаях возможного деления на ноль. Казалось бы откуда программисту знать какую конкретно опцию выставить, но кошка обычно знает чьё мясо съела.

Сама библиотека `UBsan`, поддерживающая всю эту функциональность перекочевала в GCC из LLVM/Clang поэтому доступна и там.

Очень полезно прогонять ваши программы под санитайзерами на предмет поиска сложных случаев UB которые вы не всегда сможете проверить глазами.

**Домашняя наработка:** попробуйте под санитайзером следующую программу на языке C:

```

1 #include <stdlib.h>
2
3 int a = 0xecfb39f5;
4 unsigned short *d = (unsigned short *) &a;
5
```

```

6 int main () {
7     if ((unsigned long) (65536 * d[1] + d[0]) < (1UL << 28))
8         abort ();
9 }
```

Демонстрирует ли она неопределенное поведение? Если да, то где?  
Предложите как исправить ситуацию.

### 1.2.5 Статическая верификация кода

*Beware of bugs in the above code;  
I have only proved it correct, not tried it*

– Donald E. Knuth

Санитайзеры хороши, но у них есть проблема в некоторой тяжеловесности: код, обвешанный всеми санитайзерами будет исполняться недопустимо медленно. К тому же санитайзеры часто убивают воспроизведение многопоточных ошибок.

Гораздо лучше найти ошибку совсем рано, в идеале даже до компиляции. Это позволяют средства статического анализа кода. Их существует довольно много: как платные и профессиональные, такие как coverity, klocwork, pvs-studio, так и бесплатные open-source решения такие как clang-tidy.

Проверка может работать простым поиском распространенных паттернов ошибок

```

1 for (float x = 0.1f; x != 1.0f; x += 0.1f) {
2     // . . . .
```

Приведенный выше код вполне легальный и обычные компиляторы не выдадут здесь никакого предупреждения. Проблема в том, что цикл скорее всего будет бесконечным. Да и вообще использовать плавающую точку для условия цикла это плохо “пахнущая” практика.

Статический верификатор здесь выдаст нечто вроде:

```
> clang-tidy.exe tt1.c -header-filter=.* -checks=* -- -I.
2 warnings generated.
tt1.c:3:4: warning: Variable 'x' with floating point type
'float' should not be used as a loop counter
```

```
[clang-analyzer-security.FloatLoopCounter]
    for (float x = 0.1f; x != 1.0f; x += 0.1f) {} // warn
    ^
tt1.c:3:4: note: Variable 'x' with floating point type 'float'
should not be used as a loop counter
    for (float x = 0.1f; x != 1.0f; x += 0.1f) {} // warn
    ^
tt1.c:3:38: warning: loop induction expression
should not have floating-point type [cert-flp30-c]
    for (float x = 0.1f; x != 1.0f; x += 0.1f) {} // warn
```

Выдача взята из-под последнего clang-tidy. Тут в общем всё ясно, предупреждения вроде “loop induction expression should not have floating-point type” не просто говорят сами за себя, они, в некотором роде, кричат.

Использовать статическую верификацию это очень хорошая практика. К сожалению, поиск UB с помощью статических верификаторов требует довольно хорошо ориентироваться в их выдаче. Например для кода приведенного в конце прошлого раздела (см. 1.2.4) выдача будет до некоторой степени помогать, конечно, но больше запутывать.

**Домашняя наработка:** попробуйте упомянутую программу под статическим верификатором

### 1.3 Поведение, зависящее от реализации

Зависящее от реализации (*implementation-defined*) поведение языковых конструкций считается безопасным, а код с его использованием считается конформным. Несмотря на это, определенное количество опасностей подстерегает и там. В основном, эти опасности связаны с неочевидными случаями, которые могут непредсказуемо “плыть” от реализации к реализации.

Рассмотрим некоторую (очень условную) структуру, содержащую информацию о системном регистре: 26 бит его id и 6 бит его номера.

```
1 typedef unsigned sm_sysreg_t;
2
3 typedef struct SrIDR {
4     sm_sysreg_t tid: 26;
5     sm_sysreg_t inum: 6;
6 } SrIDR;
```

Пока что все хорошо. Но ровно до тех пор, пока не делается попытка использовать эту структуру для реальной сериализации битовых полей в регистре:

```

1 unsigned idr;
2 SrIDR * idrp = (SrIDR *)&idr;
3
4 idrp->tid = 0x2;
5 idrp->inum = 3;
6
7 assert (idr == 0xc000002); /* ORLY? */

```

В стандарте (C11 6.7.2.1/10) есть формулировка:

“The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined”

Это значит, что, в зависимости от реализации компилятора, в `idr` может быть как значение `0xc000002` так и `0xb3` – совсем не то, чего здесь ожидал программист.

Самое плохое в таких случаях – у вас всё будет работать. А потом в один прекрасный день вам понадобится перенести ваш проект на новый компилятор – и до свидания.

**Вопрос к студентам:** видите ли вы какие-нибудь ещё проблемы в этом коде?

## 1.4 C++ это конгломерат языковых возможностей

При изучении C++ необходимо понимать, что C++ не является монолитным языком (как C), а является сложным, исторически сформировавшимся конгломератом возможностей, зачастую порождающих взаимоисключающие стили решения конкретных задач. Кроме того, язык имеет много “почти стандартных” библиотек и популярные расширения на конкретных компиляторах. Самые известные это GNU расширения для C и C++.

Поэтому всегда, когда вы пишете на C++, вы на самом деле пользуетесь неким его подмножеством или надмножеством, логическим или же техническим. Эти лекции охватывают (не обязательно в указанном порядке) следующие основные подмножества C++:

- Подмножество, совместимое с языком C, либо расширяющее его, но не концептуально. Скажем ссылки, конечно, отсутствуют в C, но в принципе их можно отнести к этому же подмножеству. В основном этим средствам посвящена глава 2
- Средства поддержки ООП: классы, наследование, виртуальные функции, семантика копирования и перемещения. Этому посвящена глава 3.
- Шаблоны, включая метапрограммирование на шаблонах и constexpr выражения рассмотрены в главе 4
- Стандартная библиотека, исторически сама по себе представляющая сложный конгломерат изучается в главе 5
- Многопоточность, появившаяся в языке сравнительно поздно и являющаяся источником сравнительно большого количества вопросов и проблем, рассматривается в главе 6

Кроме того, есть несколько крупных тем, не составляющих подмножеств языка, но важных самостоятельно. Это:

- Вывод типов и работа с типами, включая хинты вывода типов, появившиеся в C++17
- Лямбда-функции
- Исключения, включая их влияния на проектирование (так называемые “вопросы безопасности исключений”)

Для успешной разработки и поддержки кода на C++, каждую из его частей необходимо знать (хотя бы поверхностно), а взаимодействие его подсистем необходимо понимать (и желательно глубоко). Сложность изучения языка C++ делает абсурдной задачу уложить его в любое ограниченное количество лекций, но при достаточной работе дома, это должно дать хороший старт.

Задания на домашнюю работу в этих лекциях встречаются в тексте лекций, помеченные как **Домашняя наработка**. Обычно эти задания требуют всего лишь понимания прочитанного текста или поставить некоторый эксперимент который поможет в понимании. Более сложные задачи (впрочем без какого-нибудь ранжирования сложности) сведены

в конце соответствующих разделов. Их, обычно, предваряет список контрольных вопросов, необходимых, чтобы ещё раз окунуть взглядом прошедшее. Большинство контрольных вопросов также использовалась мной как основные или дополнительные вопросы в экзаменационных билетах.

## 1.5 Домашняя наработка по стандартам и неопределенному поведению

### Контрольные вопросы:

1. Какой нормативный документ регулирует разрешение спорных вопросов относительно синтаксиса и семантики языка C++
2. В чем отличие unspecified behavior от undefined behavior?
3. Обязан ли компилятор сообщать (предупреждением или ошибкой) о случаях неопределенного поведения в вашем коде?
4. Является ли сдвиг вправо на отрицательное значение неопределенным поведением?
5. Какие случаи неопределенного поведения легко поймать компилятором и сложно санитайзером?

### Задания:

1. Найти, скачать и бегло посмотреть стандарты о которых шла речь на этой лекции. В дальнейшем необходимость консультироваться с этими документами будет периодически возникать.
2. Пусть дан код на С:

```
1 foo(const char **p) { }
2
3 main(int argc, char **argv) {
4     foo(argv);
5 }
```

**Задача:** охарактеризовать этот код с точки зрения стандарта C99, указать причины, по которым могут возникнуть проблемы.

Попробуйте проделать же самое для прочих известных вам стандартов языков С и C++

3. Пусть дан код на С:

```
1 int foo(int x) {
2     int a = (x += 0) + 3;
3     return a;
4 }
```

**Задача:** охарактеризовать этот код с точки зрения стандарта C99, указать причины, по которым могут возникнуть проблемы.

4. Пусть дан код на С:

```
1 int bar(int x) {  
2     int a = x +++++ x;  
3     return a;  
4 }
```

**Задача:** охарактеризовать этот код с точки зрения стандарта C99, указать причины, по которым могут возникнуть проблемы.

Попробуйте также C++98, сравните результат.

5. Пусть дан код на С:

```
1 int buz(int x)  
2 {  
3     int b = bar(b += foo(b));  
4     return b;  
5 }
```

**Задача:** охарактеризовать этот код с точки зрения стандарта C99, указать причины, по которым могут возникнуть проблемы.

Попробуйте также C++98, сравните результат.

6. Пусть дан код на С:

```
1 char *fst() {  
2     char *p = "wikipedia";  
3     p[0] = 'W';  
4     return p;  
5 }
```

**Задача:** охарактеризовать этот код с точки зрения стандарта C99, указать причины, по которым могут возникнуть проблемы.

Попробуйте также C++11, сравните результат.

7. Пусть дан код на С:

```
1 void snd(int *a, int i) {  
2     a[i] = i++;  
3 }
```

**Задача:** охарактеризовать этот код с точки зрения стандарта C99, указать причины, по которым могут возникнуть проблемы.

Попробуйте также C++11, сравните результат.

В основном в этих лекциях рассматривается стандарт C++14. Некоторое внимание уделено совместимости со старыми версиями и обзору новых возможностей.

# Глава 2

## Корни, кровавые корни

Язык C++ невозможно ни понять, ни оценить, если начинать осваивать его с его высокоуровневых возможностей (как иногда рекомендуют делать). Так или иначе, но C++ это не Java и программирование на нем без понимания того, что происходит под капотом, чревато крайне неприятными сюрпризами. Этот раздел будет посвящён C-подмножеству языка C++.

### 2.1 Простые задачи для языка С

*Talk is cheap. Show me the code*

– Linus Torvalds

Прежде чем начинать серьёзный разговор о C++, необходимо вспомнить простейшие конструкции языка С такие, как условие, цикл, объявление переменной, прочее. Проще всего вспомнить их на практике.

#### 2.1.1 Простые конструкции в простых программах

Например для практики хорошо подходит задача поиска N-ного простого числа.

Для целых чисел простыми являются несократимые (те которые делятся только на 1 и самих себя). В более общем случае простое число это такое, для которого из факта, что на него делится произведение всегда следует, что на него делится один из сомножителей.

Есть много способов проверить простоту числа. Наиболее очевидный из них базируется на прямом переборе делителей.

$$P(n) = \frac{n^2 - \sum_{d=2}^{\sqrt{n}} \left[ \frac{d - (n \% d)}{d} \right]}{n^2}$$

**Задание в аудитории:** написать реализацию на С.

Из явной формулы для  $P(n)$ , сразу следует способ найти N-ное простое число:

$$p_n = \min\{m < 2^n \mid \sum_{k=2}^m P(k) = n\}$$

Хорош ли этот способ? Не очень. Он предполагает чуть менее чем квадратичную сложность проверки, завернутую в линейный перебор. Это очень долго даже для 10000-го простого числа.

Впрочем, он достаточно хорош для однократного поиска. Для многократного поиска предлагается построить вспомогательную структуру данных, известную как решето Эратосфена. С каждым числом начиная с 2, необходимо связать признак: простое оно или нет. Изначально все числа пометить как простые. Потом проходя каждое простое число, пометить как составные все, которые на него делятся.

**Задание в аудитории:** написать реализацию на С.

**Домашняя наработка:** сделать решето с побитовым хранением для признака простоты. Сколько оно займет для первого миллиарда простых чисел?

Если ли лучшие методы проверки числа на простоту? Да есть. Для продвинутых студентов предлагается реализовать детерминированный тест Миллера-Рабина.

Все эти примеры следует разобрать: объявление функций и переменных, циклы, условия, вызов функций стандартной библиотеки. Для дальнейшего чтения критично важно, чтобы сохранялось понимание базовых вещей.

## 2.1.2 Структура программ и заголовочные файлы

После реализации проверки на простоту (непосредственной или решета), возникает логичная идея отделить её от основного кода и многократно переиспользовать в разных приложениях. На рисунке 2.1 показана схематичная программа, состоящая из двух модулей: nthprime.c это

основная программа, которая ищет N-е простое число и primes.c это модуль, экспортирующий три функции: `init_sieve`, `is_prime` и `free_sieve`. Для того, чтобы сообщить основной программе о внешних функциях, используется **заголовочный** файл `primes.h`. При этом текст этого файла включается через `#include "primes.h"` в оба файла с исходниками.

В этой схеме на этапе компиляции в ассемблер, будет сгенерирован ассемблерный файл `nthprime.s` содержащий вызовы этих функций и ассемблерный файл `primes.s`, содержащий их определения.

Далее линкер связывает оба этих файла в один бинарный файл, разрешая все ссылки и выдавая ошибки там, где вызвана функция, которой нет ни в одном известном модуле.

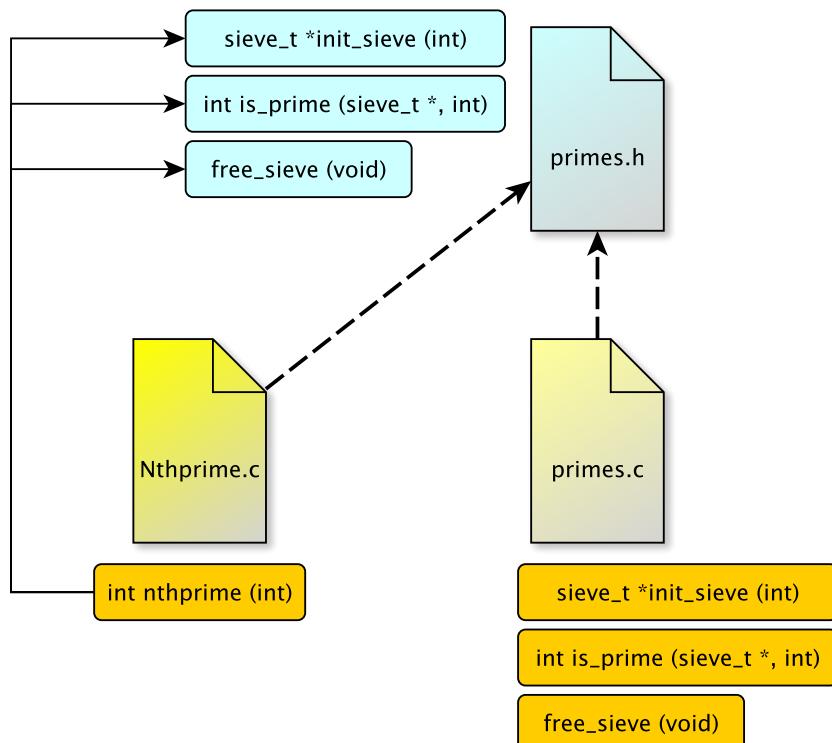


Рис. 2.1: Модульная структура программы

Если модуль `primes.c` достаточно стабилен, нет нужды каждый раз перекомпилировать его. Достаточно один раз скомпилировать его в специальный формат библиотечного файла – `libprimes.a` или `libprimes.so` после чего подключать к линкеру опцией `-lprimes`. Но это уже платформенно-специфично, в windows речь идёт о DLL, в Linux может потребоваться PIC-код для so-модулей, etc. Всё это выходит за рамки курса, но доста-

точно легко осваивается по мануалам к вашей системе.

**Вопрос к студентам:** так ли нужны h-файлы? Что если непосредственно написать объявление `init_sieve` в точке использования и предоставить линкеру самому найти тело этой функции?

**Домашняя наработка:** попробовать слинковать `libprimes` как статическую и как динамическую библиотеку и подключить к компиляции

### 2.1.3 Дьяволы деталей синтаксиса C

*Probably the oddest aspect of C  
is the declaration syntax*

– Dennis Ritchie

В эпиграф вынесены (с некоторым сокращением) слова создателя языка C, сказанные им в интервью журналу C++ Report в 2000-м году, после нескольких десятков лет торжественного шествия языка по миру. И, в общем, человек знал что говорил. Пусть стоит задача прочитать простое объявление на языке C

```
1 char *ap[20];
```

**Вопрос к студентам:** что это?

**Вопрос к студентам:** как правильно написать указатель на массив?

Теперь рассмотрим модификатор `const`.

**Вопрос к студентам:** есть ли разница между следующими объявлениями:

```
1 const int *xc1;
2 int const *xc2;
3 int * const xc3 = 0;
```

**Вопрос к студентам:** как правильно написать константный указатель на константные данные? Сделать это надо двумя способами, не меньше.

**Вопрос к студентам:** что значит ключевое слово `volatile`, каковы способы его использования?

**Вопрос к студентам:** Запишите теперь константный указатель на волатильные данные. Имеет ли он смысл? Можете ли вы представить

ситуацию, когда вам может понадобиться волатильный указатель на константные данные?

Кроме того константность и волатильность также могут комбинироваться. Скажем возможен `const volatile` указатель на `const volatile` данные. В стандарте и прочих нормативных документах константность и волатильность любят объединять под общим названием “cvqualifier” (C++14 3.9.3). Далее также будет употребляться термин “свквалифицированный тип”, это значит тип, к которому (возможно) добавлена константность, волатильность или обе сразу.

**Вопрос к студентам:** чем отличается `const volatile int` от `const int`?

Может показаться, что это создаёт некий комбинаторный взрыв типов: у нас могут быть константные или волатильные указатели на другие указатели либо на данные, тоже с cv-квалификаторами. И это действительно так и даже хуже.

#### 2.1.4 Чтение объявлений, как источник радости

**Вопрос к студентам:** прочитайте на русском языке следующее объявление:

`1 char* const *(*next)();`

Алгоритм чтения таких объявлений (приведён например в [2]) следует из стандарта и в данном случае довольно прост:

- Идём к имени переменной “next”
- Группируем её с содержимым её скобок: “next is a pointer to”
- За пределами скобок объявление функции имеет больший приоритет, поэтому “next is a pointer to a function, returning”
- Далее обрабатываем слева звёздочку, имеем: “next is a pointer to a function, returning pointer to”
- И далее парсим `char * const`: “next is a pointer to a function, returning pointer to constant pointer to character”

**Домашняя наработка:** написать программу, которая парсила бы произвольное объявление на С. Также учесть возможность агрегатных

типов: `enum`, `struct`, `union`. Проверьте свою программу против онлайн-сервиса [cdecl.org](http://cdecl.org)

**Вопрос к студентам:** с учетом новых знаний, прочитать самостоятельно

```
1 char *(*c[10])(int **p);
```

### 2.1.5 Ваш друг `typedef`

Рассмотрим прекрасное объявление типа:

```
1 void (*signal(int sig, void (*func)(int)) ) (int);
```

С помощью `typedef` его можно переписать, доставив гораздо меньше боли глазам читающего (и без риска посадить случайную опечатку).

```
1 typedef void (*ptr_to_func) (int);
2 ptr_to_func signal(int, ptr_to_func);
```

Умение читать запутанные объявление не означает необходимости их писать (если вы не участвуете в международном конкурсе по обfuscации программ). Где возможно, следует использовать `typedef`, это хороший стиль. Но хороший личный стиль (с другой стороны) это не основание не иметь навыка чтения запутанных объявлений. Каждый программист в жизни имеет дело с тоннами унаследованного кода, где встречается всякое.

У `typedef` есть некоторые опасности, которые часто ускользают от новичков. Так например `typedef` может объявить сразу несколько синонимов типов:

```
1 typedef int *ptr, (*fun)(), arr[5];
```

Это считается дурным тоном, старайтесь этого избегать. Ещё более дурным тоном считается зарыть `typedef` вглубь объявления, что тоже позволяется синтаксисом.

```
1 unsigned const long typedef int volatile *kumquat;
```

Опытный программист на языке С часто чувствует себя свободнее с препроцессором, скажем вместо

```
1 typedef int * int_ptr;
```

Есть соблазн написать

```
1 #define INT_PTR int *
```

Но здесь есть тонкая ловушка:

```
1 INT_PTR x, y; /* x is pointer, y is int */
2 int_ptr w, v; /* w, v are pointers */
```

Здесь переменные `x` и `y` будут разных типов, а `w` и `v` – одного типа.

Синтаксис `typedef` унаследован от С и в C++ производит странное впечатление. Того же эффекта можно добиться использованием `using`.

```
1 typedef int * int_ptr;
2 using int_ptr = int *; // the same but in C++ style
```

К сожалению ключевое слово `using` очень перегружено и означает кучу самых разных вещей. Подробнее о его использовании в шаблонных контекстах мы поговорим в (4.2.9).

Первое и основное отличие языка C++ от языка С заключается в том, что в C++, благодаря развитой системе типов, препроцессор нужен гораздо реже и гораздо меньше.

**Домашняя наработка:** составьте предварительный список обоснованных и необоснованных применений препроцессора в программах на C++ прежде чем продолжать чтение (2.2).

### 2.1.6 Мелкие шероховатости

С-подмножество языка C++ лишено некоторых возможностей, к которым мог быть привычен программист на С. Здесь они вкратце освещаются.

- Функция `main()` в C++ не может быть вызвана из пользовательского кода. В языке С это разрешено, хотя и несколько необычно.
- Прототипы функций обязательны в C++, но optionalны в ранних версиях С.
- Сложная и развитая система инициализации в С обычно не работает в C++

```
1 struct T {
2     union {
3         struct {
```

```

4         int x, y, z;
5     };
6     int xyz[3];
7 };
8 int a;
9 };
10
11 struct T v = { { .x = 1, .y = 2, .z = 3}, 4 };
12 struct T w = { { .xyz[0] = 1, .xyz[1] = 2, .xyz[2] = 3}, 4
13 };
14 struct T x[] = { [0].a = 1, [1].a = 2 };

```

Такой код инициализации легален в С и совершенно нелегален в C++

- Имена, определяемые через `typedef` не могут совпадать с существующими именами структур в C++, но могут в С (последний требует явной квалификации `struct`).
- При присвоении к `void *` указателю указателя на иной тип, C++ требует приведения (С не требует, но оно считается хорошим тоном).
- C++ вводит более десяти новых ключевых слов. Они могут быть использованы как идентификаторы в программе на С, но компилятор C++ выдаст ошибку.
- В языке C++ объявление переменной может появиться везде, где может быть выражение; в С, объявления должны быть в начале блока.
- Имя структуры или объединения во внутренней области видимости скроет такое же имя любой переменной во внешней области видимости в C++, но не в С.
- У символьных литералов тип `char` в C++, но тип `int` в C90. То есть `sizeof('a')` даёт 1 в C++, но может дать большее значение в C90.

## 2.2 Борьба с препроцессором

*Almost every macro demonstrates  
a flaw in the programming language,  
in the program, or in the programmer.*

– Bjarne Stroustrup

Пример с `typedef` показывает, что использование возможностей языка лучше, чем использование препроцессора. Есть целый ряд возможностей, которые кардинально отличают даже базовое подмножество C++ от чистого С и использование которых является хорошим тоном. Но есть и случаи, когда обойтись без препроцессора сложно.

### 2.2.1 От макросов к константам времени компиляции

С-стилем в программировании является использование всюду препроцессора:

```

1 #define BITS_PER_BYTE 8
2 #define sieve_type_bytes sizeof(sieve_type)
3 #define sieve_type_bits (sieve_type_bytes * BITS_PER_BYTE)
4 #define max_bit (1U << (sieve_type_bits - 1))

```

У такого подхода есть ряд недостатков: нет явного указания и контроля типов, происходит текстовая подстановка, в результате имена “портятся” для применения в программе, не пишется отладочной информации об именах констант и так далее.

В противоположность этому C++ подход состоит в явном задании хорошо типизированных констант:

```

1 const unsigned BITS_PER_BYTE = 8;
2 const unsigned sieve_type_bytes = sizeof(sieve_type);
3 const unsigned sieve_type_bits = sieve_type_bytes *
    BITS_PER_BYTE;
4 sieve_type max_bit = (1U << (sieve_type_bits - 1));

```

Удивительно, что этот код не является корректным кодом на языке С, потому что в С одни константы не могут использоваться для инициализации других констант. Кроме того строго говоря они не являются

константами времени компиляции, а определения препроцессора являются. С константностью и особенностями `constexpr` мы столкнёмся в (4.11.1).

Ещё в большей степени это касается работы с перечислениями. Она была введена в C в стандарте C99, но до сих пор автор встречает программы, активно использующие макросы.

```
1 /* C-style */
2 #define kword_TOKEN 10
3 #define kword_LITERAL 30
4 #define kword_SPACE 40
5 /* C++-style */
6 enum keyword {TOKEN = 10, LITERAL = 30, SPACE = 40};
```

Здесь кроме очевидной экономии места, вводится тип `keyword`, который может принимать только определённые в `enum` значения, что также является преимуществом.

**Вопрос к студентам:** видите ли вы какие-нибудь проблемы с таким подходом?

Кроме того, было бы очень хорошо иметь возможность в цикле пройти по всем элементам перечисления (сейчас это сложно, так как они не обязаны занимать не только последовательные, но даже и просто регулярно расположенные номера).

Более подробно эти проблемы и варианты их решения будут изложены несколько позже (см. 2.10.3)

### 2.2.2 Внешние, статические и встраиваемые функции

Любой идентификатор (переменная, функция) введенный в программе может иметь внешнее или внутреннее связывание (linkage). При разговоре о структуре программ (2.1.2) уже было упомянуто, что линкер может найти функцию из одного модуля, объявленную в другом модуле. Но если бы это было верно для всех функций, две функции с одинаковыми именами не могли бы существовать в пределах всех единиц трансляции.

К счастью, в C и в C++ функция может быть объявлена имеющей внутреннее связывание (модификатор `static`), что означает, что она не

экспортируется из модуля а используется только внутри него. Для указания внешнего связывания можно использовать ключевое слово `extern` но вообще-то оно считается для функций указанным по умолчанию.

Обратите внимание: ключевое слово `static` для переменной в глобальном диапазоне и внутри функции означает разные вещи. В глобальном диапазоне это ключевое слово означает внутреннее связывание, а внутри функции – статическую переменную (то есть сохраняющую значение между вызовами функции).

Кроме внутреннего связывания, С и С++ поддерживают для функций (но не для переменных!) спецификатор `inline`, обозначающий, что функция рекомендуется для встраивания: то есть вместо настоящего вызова, её тело будет размещено в точке вызова. Важно понимать, что использование ключевого слова `inline` это не приказ, а совет, оно увеличивает шансы на подстановку но не обязует компилятор её производить.

Увы в С++ с ключевым словом `inline` тоже не всё просто: оно влияет также на тонкие вопросы жизни объектов в многомодульных программах.

**Вопрос к студентам:** если компилятор может не подставлять функцию, обозначенную как `inline` и может подставить функцию не обозначенную таким образом, зачем вообще писать этот модификатор?

Правильный ответ: возможности компиляторов для встраивания как правило ограничены, так как слишком большой размер модулей это тоже плохо. Поэтому крайне часта ситуация, когда стоит выбор между несколькими функциями, каждую из которых можно встроить, но не все вместе. В этих случаях `inline` помогает компилятору расставить приоритеты.

Благодаря встраиванию, С++ сильно выигрывает для определения небольших функций, где в С были выгодны макросы. Хороший пример – вычисление максимума двух чисел.

Типичный контекст использования

```
1 int a = 5, b = 0;
2 assert(MAX(a,b) == 5);
```

Вариант реализации на С:

```
1 #define MAX(a, b) ((a) > (b) ? (a) : (b))
```

**Вопрос к студентам:** Зачем столько скобок?

Вариант реализации на C++:

```
1 inline long max(long a, long b) {
2     return (a > b) ? a : b;
3 }
```

Оба эти варианта работают для тривиальных случаев. Ниже приведён чуть более сложный код, для которого начинаются проблемы.

```
1 int a = 5, b = 0;
2 int c = MAX(++a, b);
3 int d = MAX(++a, b+10);
```

**Вопрос к студентам:** что будет содержаться в `c` и в `d`? А что если подставить вызов функции `max`?

Пока что синый вариант, тем не менее, обладает некоторой притягательностью: если исключить сайд-эффекты, то во-первых он является безразличным относительно типов аргументов (type generic), так что его можно использовать и для `double` в то время, как плюсовый вариант как он сейчас написан, требует только чисел типа `long`. В (2.7.3) будут рассмотрены средства, которые позволяют написать обобщенный максимум на C++

Есть ещё аргумент относительно штрафа на производительность из-за вызова функции. Но поскольку вызов `max` почти всегда будет проинлайнен, эффективность C++ метода как минимум не страдает. Более того, почти всегда, когда в С использовался `void*` для передачи параметров неопределенного типа, в C++ можно написать более эффективный код, используя шаблоны. Подробный разбор этого следует несколько отложить (см. 2.7.3).

Удивительно, но даже без встраивания функций, замена макросов на функции может дать прирост производительности

```
1 #define SQR(x) ((x)*(x))
2 unsigned sqr (x) { return x*x; }
```

в контексте использования

```
1 unsigned t = SQR(f(x));
```

Если в квадрат возводится результат вычисления функции `f`, то вариант с макросом вызовет её дважды, а вариант с функцией вызовет её один раз.

Модификатор `inline` появился и в языке С (начиная с С99) но там он работает по другим правилам: функция, объявленная `inline` по умолчанию считается не `static` а `extern`. Внешние встраиваемые функции означают, что в каждом модуле вы должны предоставить тело такой функции для возможного встраивания, но, если встраивания не произойдёт, будет вызвана та из них, где `extern` указан явно.

На C++ `extern inline` тоже можно писать, но тут правила гораздо мягче: стандарт всего лишь требует от линкера выбрать какую-то одну реализацию чтобы сгенерировать из неё тело для вызова (C++14 7.2.1).

**Вопрос к студентам:** в чём по-вашему может быть смысл писать `extern inline` функции?

### 2.2.3 Когда нужен препроцессор

В некоторых случаях препроцессор всё-таки нужен. Так, например, в (2.1.2) был описан случай включения заголовочных файлов с использованием `#include`. Возможно в стандарте C++20 пройдет предложение по модулям в языке и тогда эта практика станет устаревшей, но пока это единственная (и хорошо проверенная временем) возможность организовать модульность.

Также почти всегда нужны так называемые стражи включения, позволяющие избежать включения одного и того же модуля дважды. Например в ситуации когда в процессе включения возникает циркулярная ссылка, они бывают полезны чтобы прервать её. Но и вообще простановка стражей включения это хороший тон, так что реальный модуль `primes.h` из (2.1.2) будет выглядеть как-то так:

```

1 #ifndef PRIMES_GUARD_
2 #define PRIMES_GUARD_
3
4 /* ... something meaningful ... */
5
6 #endif

```

Ещё один случай это отключение неиспользуемого кода по опции сборки

```

1 #ifndef VERBOSE
2 #define VERBOSE 0
3 #endif

```

```

4
5 #if (VERBOSE == 1)
6 /* ... some logging ... */
7 #endif

```

Эта программа скомпилированная с `-DVERBOSE=1` будет выполнять дополнительный код, заключенный под директиву препроцессора.

Иногда нужно использовать черную магию препроцессора, например возможности по конкатенации:

```

1 struct command
2 {
3     char *name;
4     void (*function) ();
5 };
6
7 struct command commands[] =
8 {
9     { "quit", quit_command },
10    { "help", help_command },
11    /* . . . . */
12 };

```

Гораздо яснее и проще

```

1 #define COMMAND(NAME) { #NAME, NAME ## _command }
2
3 struct command commands[] =
4 {
5     COMMAND (quit),
6     COMMAND (help),
7     /* . . . . */
8 };

```

Это уберегает от человеческих ошибок (скажем несоответствия имени хендлеру), но использование таких конструкций должно быть исключением, а не правилом.

Иногда применение препроцессора связано с необходимостью трюков для применения по косвенности:

**Вопрос к студентам:** что ниже следует поставить, чтобы добиться искажения функции по значению переменной:

```
1 #define VARIABLE 3
2
3 /* ... some magic? ... */
4
5 extern void NAME(mine)(char *x);
6 /* creates mine_3 function */
```

При использовании препроцессора важном понимать, что он работает в один проход.

```
1 #define h_h # ## #
2 #define mkstr(a) # a
3 #define betw(a) mkstr(a)
4 #define join(c, d) betw(c h_h d)
5 char p[] = join(x, y);
6 cout << p << endl;
```

**Вопрос студентам:** что на экране?

Ответ в общем очевиден: на экране "**x ## y**" причём символ **##** получился после конкатенации двух символов хеша **# ## #**.

Допустим, теперь мы хотим, чтобы эта конкатенация исполнилась до превращения в строку и на экране было "**xy**". Здесь добавление лишнего уровня косвенности уже не поможет. Увы, увы, препроцессор не начинает проверку на возможную идентификацию аргументов для свежеобразованных токенов. Именно поэтому, к слову, он не Тьюринг полон. Тьюринг-полный, мощный и правильный препроцессор называется GNU M4 (<https://www.gnu.org/software/m4/m4.html>) но, если честно, он мало ком всерьёз используется.

**Домашняя наработка:** в качестве упражнения написать свой многопроходный препроцессор.

Все рассмотренные в этом разделе случаи полезного использования препроцессора, касаются, как можно заметить, его базового применения: обработки текста программы. Во всех остальных случаях при программировании на C++ использования препроцессора стараются избегать.

## 2.3 Базовые концепции языка

Есть существенное отличие между знанием как работает язык и знанием **почему** он так работает. Стандарты языков и руководства по программированию часто написаны в очень разных терминах. В основе C++ лежит целая совокупность фундаментальных понятий: точки объявления и инстанцирования, полные и неполные типы, правые и левые ссылки, области видимости и спецификаторы связывания. Все они крайне важны. В этом разделе будет рассмотрено некоторое количество таких корневых концепций, без которых дальнейшее понимание языка будет затруднено.

### 2.3.1 Объявления и определения

До сих пор действия с объявлениями и определениями выполнялись интуитивно, без полного понимания того, что это такое. Настало время обратить внимание на детали.

Объявление (declaration) это введение идентификатора и описание типа.

```

1 extern int bar;
2 extern int g(int, int);
3 double f(int, double); // extern can be omitted for function
    declarations
4 class foo;           // no extern allowed for class
    declarations

```

Нужно обратить внимание на перегруженный смысл ключевого слова **extern** в этом контексте. В (2.2.2) оно использовалось для задания внешнего/внутреннего связывания, тогда как здесь используется оно же, но для указания, что переменная объявлена, но не определена.

Объявление достаточно компилятору, чтобы разрешить ссылки на данный идентификатор.

Определение (definition) это реализация типа или выделение памяти объекту.

```

1 int bar;
2 int g(int lhs, int rhs) {return lhs*rhs;}
3 double f(int i, double d) {return i+d;}
4 class foo {};

```

Определение достаточно линкеру, чтобы реализовать идентификатор в объектном коде.

Объявление может встречаться сколько угодно раз. Определение может встретиться лишь один раз. Это называется One Definition Rule (ODR) и оно должно чётко соблюдаться. Нарушение ODR – UB. При этом любое определение также является объявлением.

У ODR есть несколько уровней строгости и из него есть несколько исключений, подробнее см. (C++14 3.2) где они перечислены.

- функции и переменные с внешним связыванием должны быть определены один раз на все единицы трансляции
- встраиваемые (inline) функции, статические (static) функции и переменные, определения пользовательских типов, шаблоны классов и их частичные специализации должны быть определены не более одного раза в каждой единице трансляции

Можно запомнить мнемоническое правило чтобы не путать объявление с определением, а в англоязычной литературе declaration и definition, нужно смотреть на словарное упорядочение. Буква “б” расположена по алфавиту раньше, чем “п”. Значит в словаре слово “объявление” будет раньше, чем “определение” и так же declaration в английском словаре идёт раньше, чем definition. И так же в программе – объявление всегда должно идти раньше определения.

Не стоит путать также определение переменной и её инициализацию. Переменная может быть определена но не инициализирована. В С и в C++ чтение неинициализированной переменной это UB (например C99 6.3.2.1/2).

Важной концепцией является точка объявления (Point of Declaration, сокращенно PoD). Объявление имени считается законченным когда записан полный идентификатор (то есть **до** возможных инициализаторов). Поэтому:

```

1 int x = 2;
2 {
3     int x /* PoD */ = x;
4 }
```

В этом случае значение внутреннего x не определено. Но в следующем случае:

```

1 int x = 2;
2 {
3     int x[x] /* PoD */;
4 }
```

всё корректно и вложенный массив имеет два элемента.

**Вопрос к студентам:** хорошо, в первом случае оно не определено. Но если программа будет, тем не менее, читать значения из такой неинициализированной переменной, будет ли каждый раз получаться одно и то же (пусть мусорное) значение или могут быть всегда разные?

Интересно, что строчка одного и того же вида может служить определением или объявлением в зависимости от контекста:

```

1 typedef void T();
2 T t; // declaration of function "t"
3
4 typedef int T;
5 T t; // definition of object "t"
```

Эта асимметрия связана с (исторической) асимметрией объявлений функций и переменных.

Также при определении не должен сужаться объявленный linkage specifier, поэтому вот так можно:

```

1 static int x;
2 extern int x; // x still static
```

А вот так нельзя:

```

1 extern int x;
2 static int x; // compilation error
```

Во втором случае storage class переменной уже заявлен как внешний и не может быть переопределён.

Больше про объявления и определения классов в (3.1.9).

### 2.3.2 Область видимости и время жизни

С концепцией точки объявления, связана концепция области видимости. Область видимости переменной, объявленной внутри блока из

фигурных скобок это текст программы от точки определения до закрывающей фигурной скобки блока. В стандарте можно посмотреть правила для других областей видимости (C++14 3.3).

Гораздо более интересной концепцией является время жизни переменной. Время жизни переменной бывает не равно её области видимости.

Область видимости (scope) это те места в программе, где возможно обращение к переменной.

Время жизни (lifetime) это все то время, когда возможно **корректное** обращение к переменной.

Чтобы не сравнивать время с пространством, можно также сказать, что кроме области в которой переменная видима, есть область в которой она валидна. Но иногда такое упрощение искажает смысл, так что лучше им не пользоваться.

Для автоматических и нелокальных статических переменных, их время жизни почти всегда совпадает с временем выполнения программы, проведённом в их области видимости.

**Вопрос к студентам:** охарактеризуйте следующий код:

```
1 int foo(int x) {
2     int y, *p;
3
4     {
5         int z = 5;
6         p = &z;
7     }
8
9     y = *p;
10    return y;
11 }
```

**Вопрос к студентам:** теперь охарактеризуйте следующий код:

```
1 int foo(int x) {
2     int y, *p;
3
4     {
5         static int z = 5;
6         p = &z;
7     }
8 }
```

```

9     y = *p;
10    return y;
11 }
```

Вещи становятся сложнее, когда в игру вступает динамическая память:

```

1 int* f (int n) {
2     int *p = malloc (sizeof(int) * n); /* 1 */
3     return p;
4 } /* 2 */

5
6 int main () {
7     int *q = f (10);
8     free (q); /* 3 */
9 }
```

Здесь область видимости переменной `p` заканчивается в точке 2, а время жизни объекта в динамической памяти – в точке 3.

С указателями все ещё интереснее: оказывается само время жизни указателя зависит от жизни объекта, на который он указывает. Стандарт регламентирует (C99 6.4.2), что “The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime”. Например:

```

1 int *p = malloc (sizeof(int));
2 long pold = (long) p;
3 free (p);
4 assert (p == (int *)pold); /* ORLY? */
```

Здесь время жизни `p` закончилось, хотя `p` это локальная переменная и она не выходила из области видимости.

На самом деле, время жизни оканчивает не только `free` но и `realloc`. Известный технический анекдот основан на так называемом “N. Lewycky realloc” (реаллок Левицкого). Ник Левицкий, инженер LLVM compiler team предложил следующий код

```

1 int *p = malloc (sizeof(int));
2 int *q = realloc (p, sizeof(int));
3 *p = 1;
4 *q = 2;
5 if (p == q)
6     printf ("%d %d\n", *p, *q);
```

И на тогдашнем clang, этот код распечатал ему “1 2”, что, вообще говоря, безумие, потому что получается, что в программе есть два одинаковых указателя, которые указывают на разные объекты.

Конечно здесь UB из-за использования указателя после `realloc` т.е. после окончания времени жизни.

### 2.3.3 Полные и неполные типы

Если тип (структура, класс, массив) только объявлен, но не определен то этот тип считается неполным. Полным тип становится только в точке, в которой встречается его определение. Но даже неполного типа вполне хватает для объявлений. На примере структур:

```
1 struct t; /* declare type t */
2
3 extern int foo(t p); /* ok, incomplete function type */
4 extern t x; /* ok, incomplete variable */
5 extern t arr[]; /* ok, incomplete array */
```

**Вопрос к студентам:** можно ли написать определение для объекта неполного типа?

### 2.3.4 Lvalue и rvalue

Следующий пример, многим может показаться излишне простым.

```
1 x = y;
```

Что здесь написано? Здесь написано – взять адрес переменной `x` и записать по этому адресу значение переменной `y`. В этом выражении присваивания, переменная `x` находится в выражении слева, а `y` – справа и стандарт C++98 вводит специальные термины `rvalue` (right-hand-side value) и `lvalue` (left-hand-side value) интуитивно понимаемые как “нечто, что может быть справа (слева) в выражении присваивания” (C++98 3.10).

Итак, какие ограничения этот пример накладывает на `x`? Похоже, что `x` должен быть полного типа, не константным, и иметь определённое местоположение в памяти (быть адресуемым). Есть ли ограничения на `y`? Да есть. Он должен быть полного типа и этот тип должен быть совместим с типом `x` по присваиванию.

Это типичный пример того, как компилятор может решить из контекста (в данном случае из положения справа или слева от присваивания) будет ли он использовать адрес переменной или её значение в своих фактических вычислениях. Такая возможность у компилятора есть, потому что адрес переменной полного типа всегда известен во время компиляции и нет необходимости заставлять программиста его специально получать.

```
1 *(&x) = y;
```

Возможно, кстати, писать так было бы честнее.

Стандарт C++11 существенно расширяет и дополняет классификацию категорий значений, речь об этом пойдет в (3.5.1).

## 2.4 Массивы и указатели

*Should array indices start at 0 or 1?  
 My compromise of 0.5 was rejected without,  
 I thought, proper consideration*

– Stan Kelly-Bootle

Многие программисты заучивают правило для новичка: массивы в С это указатели и наоборот. Это правило, пожалуй, действительно полезно для новичка. Но в общем случае это не так. В этом разделе будут проведены чёткие разграничения.

### 2.4.1 Вспоминаем указатели

Указатель хранит адрес, по которому программа может прочитать и (в случае неконстантных данных) записать значение, того типа, из которого выведен тип указателя. Для любой переменной на протяжении её времени жизни (2.3.2) стандарт гарантирует побитовую неизменность адреса (C11 6.2.4).

С неконстантным указателем можно сделать следующее:

1. Объявить и инициализировать константой

```
1 int *x = 0;
```

2. Объявить и инициализировать адресом переменной или функции

```
1 struct str_t {int x; int y;};
2 int foo (int x);
3 int c = 2;
4 str_t s = {1, 2};
5 int *p = &c;
6 int (*pfoo) (int) = &foo;
7 str_t *ps = &s;
```

3. Присвоить иное значение

```
1 int bar (int x);
2 p = x;
3 x = &c;
4 pfoo = &bar
```

4. Разыменовать и получить или изменить значение

```
1 c = *x;
2 *p = 5;
3 c = (*ps).x;
4 c = ps->x;
5 ps->x = *p;
```

5. Использовать индексацию, похожую на массив

```
1 p[0] = c;
2 *(p+0) = c;
```

6. Возвращать из функций

```
1 int *foo (int x) { return &x; }
```

Неконстантные указатели являются lvalue. В принципе, можно записать:

```
1 int p = 3;
2 int *pp = (int *) p;
3 return *pp;
```

Этот код является дурным тоном, он будет плохо переносим и совершенно точно тут указатель используется не по назначению, но важно, что это можно сделать. Указатель это честная ячейка памяти. Он хранит то, что туда положил программист, как это показано на рисунке 2.2.

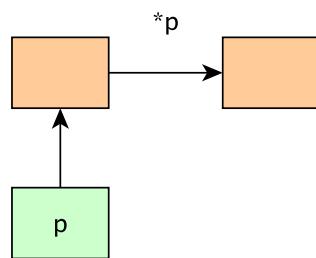


Рис. 2.2: Визуальное представление указателей

**Вопрос к студентам:** можно ли сериализовать указатели, например в файл, а потом восстанавливать и использовать?

**Вопрос к студентам:** можно ли для переменной сказать сколько указателей на неё в данный момент существует?

Это, вообще говоря, сильно снижает возможности компиляторов к оптимизации.

```
1 float *b;
2 int x;
3 // .... (etc)
4 x = 1;
5 *b = 5;
6 x = x + 3;
7 foo (x);
```

Этот код мог бы быть оптимизирован как показано ниже

```
1 x = 4;
2 *b = 5.0;
3 foo (4);
```

Но компилятор не имеет права этого делать: в строчках кода помеченных троеточием вполне могло встретиться нечто вроде утечки адреса с конвертацией типа

```
1 b = (float *) &x;
```

И в этом случае запись `*b = 5.0` непредсказуемо изменит `x`.

В связи с этим, сообществом и большинство компиляторов (но не стандартом) поддерживаются **strict aliasing rules**. Их суть – в запрещении доступа к данным одного типа по указателю другого типа. Большинство компиляторов проверяют strict aliasing по умолчанию и требуют специальных опций для его отключения.

Кроме того, в стандарте языка C99 (но не C++) для указателей поддерживается модификатор `restrict` (C99 6.7.3)

```
1 void f (int n, int * restrict p, int * restrict q){
2     while (n-- > 0)
3         *p++ = *q++;
4 }
```

Здесь программист как бы дает компилятору обещание, что на вход `f` придут непересекающиеся области данных.

### 2.4.2 Арифметика указателей

Кроме всего прочего, указатели можно складывать с целыми числами, а также вычитать из них целые числа и указатели друг из друга.

Для сложения, указателем может быть любой из операндов (но только один из них, сложить два указателя нельзя). Для вычитания обязательно левый операнд должен быть указателем, а правый – указателем или числом

```

1 int arr[] = {0, 1, 2, 3, 4, 5};
2 int a = 0;
3 int *b = &arr[0];
4 int *c = b + a;      // ok
5 int *d = a + b;      // ok
6 int *e = c + d;      // ok
7 ptrdiff_t f = c - d; // ok
8 int *g = c - a;      // ok
9 int *h = a - c;      // error

```

Следует обратить внимание на тип `ptrdiff_t` в коде выше – это специальный тип, достаточно широкий, чтобы хранить разность указателей. Следует по возможности пользоваться им, но, как запасной вариант, есть тип `long` (которого увы не хватает для LLP64 систем, таких как Win64, но он вполне пригоден для почти всех unix-специфичных приложений). Он находится в хедере `<stddef.h>`.

**Вопрос к студентам:** как вы думаете возможно ли посчитать разницу между указателями на стеке и в куче?

```

1 int *t = malloc (sizeof(int));
2 int k = 4;
3 ptrdiff_t d = t - &k; /* ok? */

```

### 2.4.3 Нулевые указатели

Особой разновидностью указателей являются нулевые указатели. Их смысл – сигнализировать об отсутствии указанного значения, будь то конец строки, завершение односвязного списка или не выделенная ещё память. Первое, что нужно о них запомнить: нулевой указатель может быть не нулевым. В этом смысле константы 0, `NULL` или, скажем, `nullptr` из нового стандарта это разные константы. Для конкретной архитектуры значение `NULL` может быть `0xfffff0000`, или любым другим.

Но во всех распространенных архитектурах `NULL == 0` и чтобы сохранить обратную совместимость существующего кода, стандарт определяет правила неявной конверсии 0 в `NULL` при сравнении с ним указателей (C99 6.3.2.3).

```
1 int *t = malloc(sizeof(int));
2 assert(t); // ok, means assert(t != NULL)
```

Здесь условие под `assert` будет неявно сконвертировано в правильную форму. Конечно, программист, заботящийся о стиле, сразу пишет правильно:

```
1 int *t = malloc (sizeof(int));
2 assert (t != NULL); // great
```

Но и в первом варианте нет ошибки.

**Вопрос к студентам:** но ведь ноль может быть не только вбит, но и предвычислен:

```
1 int *zerop = (int*)(x -y);
2 assert (zerop == NULL);
```

Представим, что в этом случае `NULL == 0xfffff0000` и при этом `x == y`. Сработает ли assertion?

#### 2.4.4 Действия с массивами

Встроенные массивы (также С-массивы) исторически предшествовали указателям и представляют собой простейший вид синтаксического клея – логически непрерывную область памяти, побитую на однотипные ячейки.

С массивами также возможны некоторые действия.

1. Объявить и инициализировать списком данных (все пропущенные инициализаторы – нулевые)

```
1 int a = 5;
2 int x[10] = {6, a};
```

Следует обратить внимание: неинициализированным может быть только локальный массив. Глобальные массивы всегда инициализированы нулями.

Также следует обратить внимание. Определение вида:

```
1 int wrong[]; /* boom! */
```

Это ошибка (поскольку компилятор не знает сколько памяти выделять на этот массив). Но можно использовать такой синтаксис в объявлениих

```
1 extern int wrong[]; /* ok */
```

2. Прочитать или записать значение

```
1 int t = x[0];
2 x[1] = t;
3 x[0] = x[1];
```

3. Деградировать (decay) к указателю если он используется как rvalue

```
1 int foo (int *t);
2 int *p = &x[0];
3 *p = x;           // decay
4 a = *(x + 3);   // decay
5 a = foo (x + 5); // decay
```

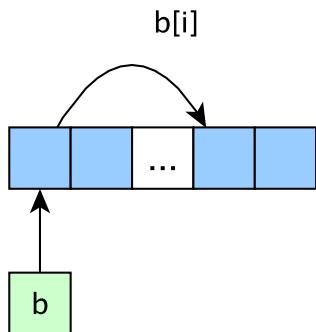


Рис. 2.3: Визуальное представление массивов

Но массивы **никогда** не употребляются вместо указателей там, где указатель это lvalue.

```
1 int v = 5;
2 int *p = &v;
3 int a[1] = {v};
4 p = a; // ok
5 a = p; // never
```

По той же причине нет возможности вернуть массив из функции:

```
1 typedef int arr_t[10];
2
3 arr_t foo () ; // Error!
```

Разумеется, невозможна никакая арифметика массивов – прибавить один массив к другому или вычесть один массив из другого в языке С невозможно.

Массивы хранят сами данные, а не указатели на них. То, что они синтаксически могут вырождаться к указателям – удобная концепция, связанная с ранней историей массивов в языке С и в дальнейшем получившая неожиданно глубокое и мощное развитие в последних версиях стандарта C++ (см. 2.11.5). Массив содержит lvalue-ячейки данных, но сам как целое он не является lvalue-ячейкой данных.

## 2.4.5 Инициализация массивов и указателей

Инициализация массивов и указателей выглядит похоже, но означает разные вещи:

```
1 char a[] = "abcdefg";
2 char *b = "abcdefg";
3 assert (a[3] == b[3]);
```

**Вопрос к студентам:** есть ли разница между этими двумя записями и какую вы предпочтёте? Почему?

Между прочим, строковые литералы в инициализации указателей это счастливое (исторически сложившееся) исключение. Строчка 2 в следующем примере не будет скомпилирована:

```
1 float a[] = {1.0, 2.0, 3.0}; // ok
2 float *b = {1.0, 2.0, 3.0}; // ka-boom!
```

**Вопрос к студентам:** есть ли разница между следующими двумя объявлениями?

```
1 int foo(int x[]);
2 int foo(int *x);
```

**Вопрос к студентам:** означает ли эта запись:

```
1 int foo(int x[16]);
```

что `foo` принимает массив из 16 символов?

#### 2.4.6 Прогулки за границами

Хорошо написанный код никогда не должен ничего считывать и записывать за границами массивов. В общем случае проблема доступа к данным вне выделенного для доступа буфера называется “buffer overflow” или “buffer underflow” и является (не только в С) одним из основных источников уязвимостей программ. Увы, такие языки как С имеют к нему родовую предрасположенность: максимальная эффективность требует отсутствия неявных проверок времени выполнения.

К чему это приводит можно проиллюстрировать на примере простейшей уязвимости:

```

1 char buff[16];
2 int pass = 0;
3
4 printf("\n Enter the password : \n");
5 gets(buff);
6
7 if (strcmp (buff, "correctpassword")) {
8     printf ("\n Wrong Password \n");
9 }
10 else {
11     printf ("\n Correct Password \n");
12     pass = 1;
13 }
14
15 if (pass) {
16     // Now Give root or admin rights to user
17     printf ("\n Root privileges given to the user \n");
18 }
```

Здесь чтобы демонстрация была эффектной придется скомпилировать без оптимизаций чтобы переменная попала на стек:

```
$ g++ p1-overrun.cpp -O0 -m32
$ ./a.out
Enter the password :
hhhhhhhhhhhhhhhh
```

**Wrong Password**

**Root privileges given to the user**

Введен очевидно неверный пароль. Но, поскольку он переполняет стек, последний символ записывается в переменную `pass` и таким образом её значение меняется и пароль засчитывается как верный.

Это самый простой, можно сказать “детский” пример такой уязвимости, в реальности всё сложнее и разнообразнее, но идея именно эта.

Частично защититься от порчи стека в GCC может помочь опция `-fstack-protector-all` – с ней программа просто упадет, оставив трассу

```
$ g++ p1-overrun.cpp -O0 -m32 -fstack-protector-all
$ ./a.out
```

```
Enter the password :
hhhhhhhhhhhhhhhh
```

```
Wrong Password
*** stack smashing detected ***: ./a.out terminated
```

Если вы используете другой компилятор, у него скорее всего существуют аналогичные средства, кроме того существует целый класс бесплатных и коммерческих программ (их хорошо искать по ключевым словам `bounds checking`).

Эти программы, конечно, тоже не панацея. В большинстве они отлавливают только уже произошедшую запись в неправильное место на стеке, реально же проблемы с уходом за границу массива могут быть гораздо более тонкими и даже вообще не включать порчи стека. Например по стандарту арифметика указателей в пределах массива не должна уходить дальше, чем на один элемент от границ массива:

```
1 int arr[] = {0, 1, 2, 3, 4, 5};
2 int *a = arr; // ok, decaying as &arr[0]
3 int *b = a + 3; // ok, *b == 3
4 int *c = a + 6; // ok, c is one past arr
5 int *d = a + 7; // undefined behavior
6 d = d - 4; // even here, *d is undefined
```

Пятая строчка здесь создает UB и весь код после этой строчки теряет предсказуемость. Даже если программа, ничего не меняя за границами массива, “честно” возвращается в допустимую область, тем не менее, UB уже случилось, жесткий диск уже отформатирован, всё пропало.

### 2.4.7 Многомерные массивы

Настоящий многомерный массив (вообще – настоящий массив) это абстрактный тип данных  $A$ , для которого определены две операции  $\text{get}(A, I)$  и  $\text{set}(A, I, V)$ . При этом выполняются аксиомы:

$$\begin{aligned} \text{get}(\text{set}(A, I, V), I) &= V \\ \text{get}(\text{set}(A, I, V), J) &= \text{get}(A, J) | I \neq J \end{aligned}$$

В качестве индекса служит кортеж произвольных объектов, обычно целых чисел.

Язык С и С-подмножество языка C++ не поддерживают семантику многомерного массива на уровне языка.

Хуже того, существует противоречие между статическими многомерными массивами (которые на самом деле одномерные с хитрым доступом) как на (рис. 2.4) и динамическими многомерными массивами (там же).

Из-за этого в точке обращения к массиву существует контекстная зависимость – неясно обращение к чему именно происходит: к jagged-вектору или к одномерному статическому массиву с двумерной адресацией?

```
1 assert(a[3][4] == *(a + 3*jmax + 4); // option 1
2 assert(a[3][4] == *((a + 3) + 4); // option 2
```

Статические массивы в памяти должны всегда быть непрерывны. Для многомерных массивов раскладка в памяти идёт построчно (см. рис. 2.4). При этом последние индексы идут последними. Так же и с инициализацией.

```
1 float flt[][][3] = {{1.0, 2.0, 3.0}, {4.0, 5.0}}; // ok
```

При этом наиболее вложенные скобки относятся к последним индексам. Первый индекс ясен из контекста и его можно опускать в инициализаторе.

Если часть вложенных инициализаторов пропущена, они считаются

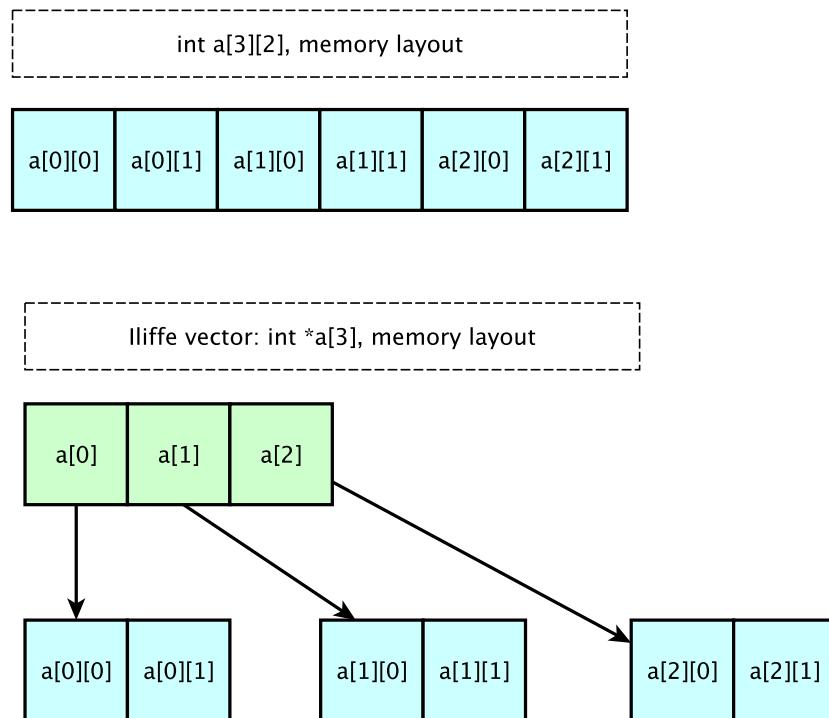


Рис. 2.4: Массивы в языке С

нулевыми, поэтому, например двумерные массивы символов могут быть проинициализированы последовательностями строковых литералов.

При передаче многомерных массивов в функции может быть опущен только первый индекс в прототипе (и он может быть любым). Остальные индексы у формального и фактического аргументов должны совпадать.

**Вопрос к студентам:** возможен ли такой прототип функции?

```
1 int func(int a[][], int n);
```

Точно так же нельзя, даже указав нужные инициализаторы, определить двумерный массив, для которого компилятор дедуцирует оба размера

```
1 int a[][] = {{1, 2}, {3, 4}}; // Error!
```

Увы, но здесь, как и в прототипах функций, должны быть указаны все размеры, кроме, может быть, первого.

**Вопрос к студентам:** в функцию можно передать массив вместо указателя. Можно ли передать двойной массив туда, где ожидается двойной указатель?

```
1 int func(int **a, int n, int m);  
2 int a[2][3];  
3 func(a, 2, 3); // huh?
```

**Вопрос к студентам:** есть ли возможность передать двойной массив туда, где ожидается одинарный указатель?

## 2.5 От указателей к ссылкам

Очень важной особенностью C++ является введение довольно низкоуровневой, но принципиально новой конструкции – ссылок. Ссылка это альтернативное имя переменной.

```

1 int x = 5;
2 int &xref = x;
3 xref += 1;
4 assert(x == 6);

```

Каждая ссылка обязательно должна быть инициализирована в точке определения. “Ссылка сама по себе” так же как “нулевая ссылка” не могут существовать. Указатель сам по себе является местом для хранения данных, ссылка – это просто имя. Поэтому ссылки не нуждаются в явном разыменовании – каждое обращение к ним это уже разыменование. Поэтому различия существенны:

```

1 int x[2] = {10, 20};
2 int &xref = x[0];
3 int *xptr = &x[0];
4 xref += 1;
5 xptr += 1;
6 assert((xref == 11) && (*xptr == 20));

```

### 2.5.1 Использование ссылок

Главный паттерн для использования ссылок это передача по ссылке входных (неизменяемых) аргументов функций, особенно если типы являются жирными:

```

1 struct T {
2     // a lot of content here
3 };
4
5 int foo(const T &a);

```

Здесь можно использовать и указатель, но тогда в теле функции его придется постоянно разыменовывать, проверять на `NULL` и так далее. Ссылка в этом отношении удобней, прозрачней и безопасней.

С другой стороны, передача изменяемых параметров по неконстантным ссылкам – дело вкуса:

```

1 int do_r (T &a);
2 int do_p (T *a); // maybe a is array?
3
4 do_r (b); // should b be changed?
5 do_p (&b);

```

Некоторые считают, что это создаёт неоднозначность в точке использования. Другие считают, что указатели также создают неоднозначность между массивом и указателем.

Не стоит передавать по ссылкам небольшие объекты. У подрастающих программистов часто чешется написать нечто вроде:

```
1 int foo(const int &a);
```

Это допустимо, но тут лучше передать `a` по значению. Внутри компилятора ссылки обычно реализуются через указатели, которые могут превосходить по размеру небольшие целые. Поэтому тут ваш выбор:

```
1 int foo(int a);
```

Просто и понятно.

Ещё один паттерн использования ссылок – задание временных имён для безымянных объектов:

```

1 int &current = big_global_array[calc_index(t)];
2 current += 1; // ok, big_global_array changed

```

В этом деле ссылки настолько удобны, потому что по стандарту они продлевают время жизни автоматических переменных:

```

1 const int &c = a + foo();
2 use (c); // ok, c holds temporary value

```

В противовес этому использование в этом контексте указателя создало бы UB.

Здесь связывание константной ссылки (const reference binding) продлевает время жизни временного объекта. Обратите внимание: ссылка связана с временным объектом. При попытке использовать там неконстантную ссылку ничего работать не будет. Зато константная ссылка продлевает всё что угодно, даже константу

```
1 const int &lv = 0; // ok
```

Теперь `lv` это второе имя для нуля.

**Вопрос к студентам:** как вы думаете, можно ли вычислить адрес временного объекта на который ссылается `lv`?

Ссылки дают важную возможность наконец-то передавать в функцию встроенные массивы фиксированного размера с контролем длины

```

1 int foo(int arr[3]); // really: int *arr
2 int bar(int (&arr)[3]); // ok, ref to arr
3
4 int wrong[5];
5 foo(wrong); // ok
6 bar(wrong); // fail

```

Без ссылок у нас была только псевдо-возможность указать размер (ну и конечно перегрузка ключевого слова `static` в языке С для статического хинта такого рода, но вот об ЭТОМ лучше сразу забыть).

## 2.5.2 Висячие ссылки

Итак, ссылка это второе имя объекта. Но что если срок жизни объекта истёк?

```

1 int *t = malloc (sizeof (int) * 10);
2 int &r = t[2];
3 free (t); /* end of lifetime */
4 use (r); /* BOOM */

```

В этом случае ссылка считается “повисшей” (`dangling`) и любое её использование это UB.

Аналогично ссылка может повиснуть если возвращать из функции ссылку на временный объект. Последние версии современных компиляторов выдают на эту тему предупреждение, но закладываться на это не стоит.

```

1 int& refret (int x) {
2     return x;
3 }
4
5 int &v = refret (3);
6 printf ("%d\n", v); /* BOOM */

```

Здесь тоже всё очень плохо.

**Вопрос к студентам:** как исправить приведенный выше код и всё-таки вернуть валидную ссылку из функции?

**Вопрос к студентам:** что можно сказать насчёт возврата временного объекта в ссылку по значению?

```

1 int xret (int x) {
2     return x;
3 }
4
5 const int &lv = xret (1);

```

### 2.5.3 Когда ссылки уступают указателям

В целом при программировании на C++ следует повсюду предпочитать использование ссылок использованию указателей.

Но есть нюансы. Ссылки иммутабельны – назначив ссылку “другим именем” чего-то, её нельзя перевязать на что-то другое

```

1 int a = 2;
2 int b = 3;
3
4 int *pa = &a;
5 pa = &b;      // ok, now pa == &b
6
7 int &ra = a;
8 ra = b;      // ok, now a == 3, ra still means a

```

Вторая строчка в случае ссылок – не перевязывает ссылку, а присваивает значение тому lvalue на которое она ссылается. Поэтому ссылки нельзя использовать для создания динамических структур данных, таких как стеки, очереди, деревья.

Кроме того, у ссылок нет аналога (`void *`) для передачи идиомы “ссылки на неопределённый тип”. Появившиеся в новом стандарте `auto` & всё таки должны быть разрешены на этапе компиляции, что бывает невозможно:

```

1 void *read ();
2
3 // using somewhere below
4

```

```
5 int a = *(int *) read(); // read int
6 double d = *(double *) read(); // next read double
```

**Домашняя наработка** найдите другие случаи, в которых использование ссылок невозможно или нецелесообразно.

## 2.6 Искажение имён и его последствия

*If names be not correct, language is not in accordance  
with the truth of things*

– Confucius

Язык С предоставляет достаточно сильную гарантию того, что любое имя, использованное в вашей программе будет уникально связано с типом, областью видимости и так далее.

Обычно это приводит к тому, что имя отображается в ассемблер целивой машины один к одному, без искажения или с несущественным искажением (скажем к stdcall именам под Windows добавляется подчеркивание в начале, то есть `foo` станет в ассемблере `_foo`).

Язык C++ такой гарантии не даёт. Вместо этого он согласен сделать работу по созданию уникальных имён (иенную другую работу) за программиста посредством встроенного искажения (манглирования) имён.

### 2.6.1 Манглирование и перегрузка

**Вопрос к студентам:** сколько функций вычисления квадратного корня вы можете назвать из стандартной библиотеки языка С?

Правильный ответ: три (7.12.7.5) `sqrtf`, `sqrt` и `sqrtl`. Три функции с разными именами понадобилось вводить потому, что они принимают аргументы разных типов.

Используя C++ вы можете написать три функции с одинаковыми именами, но различными типами:

```

1 char bar (char x) { return x; }
2 int bar (int y) { return y; }
3 long long bar (long long z) { return z; }
```

На уровне ассемблера они могут выглядеть например так:

```

_Z3barc:
....
_Z3bari:
....
```

```
_Z3barx:
```

```
....
```

Конвенции манглирования не документированы и являются зависимыми от реализации, так что закладываться на них не надо. Но грамотно использовать механизм перегрузки функций в C++ бывает очень выгодно для облегчения читаемости вашей программы.

Для большинства компиляторов C++ существуют “деманглеры” – специальные приложения, способные по искаженному имени ассемблерной метки восстанавливать сигнатуру функции. Скажем в GCC это `c++filt`. Попробуйте, скажем: `c++filt _ZN4Anls3Cfg7_DeleteEPKv`

Искажение имён позволяет больше, чем перегрузку. По сути, все средства абстракции C++, включая ООП и шаблоны возможны в этом языке только благодаря наличию в нём искажения имён. В этом разделе рассматривается только применение в перегрузках, про искажение имён методов структур см. (2.6.4).

Иногда требуется из кода на C++ сделать некую функцию или переменную (обычно входящую в интерфейс модуля) “линкуемой в C стиле” – т.е. отображаемой в ассемблер один в один. Для этого используется спецификатор линковки `extern "C"`. Им можно как аннотировать функцию или переменную, так и брать их в блоки.

```
1 extern "C" void foo(int);
2
3 extern "C" {
4     void g(char);
5     int i;
6 }
```

Так слинкованы могут быть свободные функции и глобальные переменные, но не члены классов. Две функции с такой линковкой с одинаковым именем – нарушение ODR.

**Домашняя наработка:** посмотрите как работает манглирование в вашем любимом компиляторе. Можете ли вы установить некие закономерности?

Функция не может быть перегружена по:

- Имени аргумента
- Типу возвращаемого значения

Кроме того, функции не могут быть перегружены по cv-квалификаторам

```

1 void foo (int);
2 void foo (const int); // FAIL
3
4 void bar (char *);
5 void bar (char * const); // FAIL

```

Однако это не относится к cv-квалификаторам самого внешнего типа

```

1 void foo (int&);
2 void foo (const int&); // OK
3
4 void foo (char *);
5 void foo (const char *); // OK

```

Перегрузка даёт серьёзные преимущества по сравнению с языком С. В частности, она позволяет решать некоторые вопросы более естественно. Как пример можно рассмотреть функцию `strchr`. В стандартной библиотеке C++ она задана двумя перегрузками

```

1 const char * strchr (const char * str, int c);
2     char * strchr (    char * str, int c);

```

Здесь смысл совершенно ясен: если мы ищем символ в константной строке, то и позиция будет в константной строке и наоборот.

В языке С перегрузка невозможна, и, поэтому, там сделано куда более опасно

```
1 char * strchr (const char * str, int c); // C version
```

Здесь есть гипотетическая возможность вызвать `strchr` для константной строки и вернуть неконстантный указатель, изменение по которому ведёт к UB, поэтому за этим надо постоянно следить, чтобы так не сделать. Это неудобно, опасно и хрупко.

Но кроме положительных сторон и приятных удобств, в перегрузке есть и тонкие и неприятные моменты. Например правила её разрешения.

## 2.6.2 Правила разрешения перегрузки

После того, как компилятор составил множество функций-кандидатов на разрешение перегрузки (viable set) исключив все, которые кандидатами быть не могут, разрешение происходит в следующем порядке.

1. Идеальное совпадение (включая правильный ссылочный тип)
2. Стандартные преобразования
3. Пользовательские преобразования
4. Троеточия
5. Ссылочное связывание (для неправильных ссылок)

После каждого типа преобразований сверху-вниз, получается множество возможных функций. Если это множество состоит из одной функции, будет вызвана она. Если это множество состоит из нескольких функций, будет выдана ошибка компиляции. Если это множество пусто, будет попробован следующий тип преобразований аргументов.

Предположим, в пользовательском коде стоит вызов

```
1 foo(10);
```

А чуть ранее объявлены следующие функции:

```
1 int foo(char x) { return 0; }      // 1
2 int foo(short x) { return 1; }     // 2
3 int foo(int x) { return 2; }       // 3
4 int foo(...) { return 3; }        // 4
5 int foo(int &x) { return 4; }      // 5
6 int foo(const int &x) { return 5; } // 6
```

При таком множестве перегрузки, вызов не скомпилируется, поскольку здесь есть конфликт между (3) и (6), которые подходят одинаково хорошо.

Если стереть (6), то никаких конфликтов не остаётся. Для вызова `foo (10)` точно подходит только `foo(int)` и программа возвращает 2.

Если теперь стереть (3), то будет ошибка компиляции:

```
1 int foo(char x) { return 0; }      // 1
2 int foo(short x) { return 1; }     // 2
3 int foo(...) { return 3; }        // 4
4 int foo(int &x) { return 4; }      // 5
```

Конфликт между двумя равноправными функциями (1) и (2), обе работают через один шаг стандартных преобразований. Если далее стереть

одну из них, например (2) всё снова становится норм, программа вернёт 0. Если стереть оставшуюся функцию (1), то следующей пойдёт (4).

Если убрать всё и оставить только (5), то будет ошибка компиляции: функция `foo` выиграет перегрузку, но константа не может быть связана с неконстантной lvalue ссылкой. Зато такое ссылочное связывание засчитывается как точное совпадение если вызов происходит для lvalue.

```

1 void foo(int x); // 1
2 void foo(int &x); // 2
3 void foo(const int &x); // 3
4
5 int x = 42;
6 foo(x); // (1, 2) or (1, 3) -> conflict
7           // (2, 3) -> (2)

```

Если вдуматься в эти правила они уже не выглядят столь марсианскими, правда? Хех. Это от того, что я рассказал вам их **не все**. На уровне С подмножества, всё выглядит довольно логично, но вы уже сейчас можете заглянуть в перегрузку с учётом шаблонов (см. 4.1.3)

Начиная с C++11, перегрузки можно стирать, используя ключевое слово `delete`

```

1 // for int
2 int foo (int number);
3
4 // not for char or bool
5 int foo(bool) = delete;
6 int foo(char) = delete;

```

Такой синтаксис запрещает перегрузку с заданными типами аргументов. Тем не менее, эти функции участвуют в перегрузке, поэтому при точном совпадении

```
1 foo(true); // error, not foo(int)
```

будет именно ошибка компиляции, вместо неявного приведения типа.

Перегрузка функций это очень общий механизм, к нему ещё предстоит вернуться неоднократно.

### 2.6.3 Аргументы по умолчанию

Отсутствия обязательств C++ быть близким к машине позволяет ещё более удивительные вещи, такие как аргументы по умолчанию. Это значения аргументов функции (всегда в конце списка аргументов), которые не обязаны быть указаны при вызове

```
1 char foo(char x = 0);
2 int bar(int x, int y = 5);
3 long buz(long x, long y = 5, long z = 0xafff);
```

При вызове параметры по умолчанию могут быть теперь опущены

```
1 foo(); // OK, x = 0
2 foo('a'); // OK, x = 'a'
3 buz(3, 6); // OK, x = 3, y = 6, z = 0xafff
```

Аргументы по умолчанию не входят в тип функции. Например они запрещены в указателях на функции. В терминах примера выше:

```
1 typedef char(*pfoo_t)(char);
2 pfoo_t pf = foo; // OK
3 pf(); // FAIL, no default arguments any more
4 pf('a'); // OK, foo called, x = 'a'
```

По значению аргумента по умолчанию функция также не может быть перегружена (по тем же причинам – он не входит в тип).

Параметры по умолчанию могут быть добавлены в функцию при последующих переопределениях, но не могут быть далее изменены.

```
1 char foo(char x);
2 char foo(char x = 0); // OK, adding default
3 char foo(char x = 1); // FAIL, default can not be changed
```

В качестве параметров по умолчанию могут быть использованы глобальные переменные и даже вызовы функций

```
1 int a = 1;
2 int f(int x);
3 int bar(int x, int y = f(a));
```

Теперь в программе можно изменять глобальную переменную и в качестве параметра по умолчанию `bar` будет подхвачено её значение.

## 2.6.4 Структуры в C и в C++

В языке C структура являлась способом ввести пользовательский тип, являющийся механическим объединением разнородных данных:

```
1 typedef struct pair {
2     int x;
3     int y;
4 } pair_t;
```

**Вопрос к студентам:** что за странный `typedef`, зачем он нужен?

Можно написать функцию, создающую обратную пару:

```
1 pair_t transpose_pair(const pair_t *pthis) {
2     pair_t pr = {pthis->y, pthis->x};
3     return pr;
4 }
```

**Вопрос к студентам:** обоснована ли здесь передача по указателю?

Вызвать эту функцию тоже несложно:

```
1 pair_t p = {2, 3};
2 pair_t t = transpose_pair (&p);
3 assert ((p.x == t.y) && (p.y == t.x));
```

Такие типы возможны и в C++. Но в C++ была также добавлена принципиально новая возможность группировать данные с методами их обработки внутри структуры.

```
1 struct pair_t {
2     int x;
3     int y;
4     pair_t transpose_pair ();
5 };
6
7 pair_t pair_t::transpose_pair () {
8     pair_t pr = {this->y, this->x};
9     return pr;
10 }
```

Из кода выше, в том числе, видно, что в C++ была исключена необходимость добавлять `struct` к символьному имени структуры, что делает ненужным оставшийся в C-style коде `typedef` тэга структуры на её имя.

Вызов будет выглядеть чуть иначе:

```
1 pair_t p = {2, 3};
2 pair_t t = p.transpose_pair ();
3 assert ((p.x == t.y) && (p.y == t.x));
```

Это работает за счёт искажения имён (см. 2.6). В итоговом ассемблере имена `transpose_pair` и `pair_t::transpose_pair` будут разными и у вызывающего кода не будет проблем с тем, что именно он вызывает. В итоговом ассемблере, имя метода встретится, например, в виде:

`_Z14transpose_pair4pair:`

Определённая так функция называется функцией-членом (member function), а также методом структуры. Объявление метода происходит внутри определения структуры, определение метода имеет явную квалификацию того к чему метод относится. Обратите внимание, что символ сдвоенных двоеточий такой же, как и в случае пространств имён. Он дословно означает “пространство имён, задаваемое структурой”.

Ключевое слово `this` обозначает неявный (всегда первый) аргумент в методе класса и является указателем на экземпляр структуры, для которой он вызывается. Поэтому `this->y` это поле `p.y` при вызове `p.transpose_pair()` но это будет уже `t.y` при вызове `t.transpose_pair()`

По умолчанию `this` можно опускать, чтобы не загромождать код. Имена полей класса всё равно являются идентификаторами из наиболее охватывающего пространства имён. Поэтому следующий код является и законным и совершенно эквивалентным приведенному выше:

```
1 pair_t pair_t::transpose_pair() {
2     pair_t pr = {y, x};
3     return pr;
4 }
```

Такой подход делает появление имён `x` и `y` в теле метода несколько неожиданным, поэтому многие авторы книг по хорошему стилю рекомендуют выделять название полей. Саттер и Александреску [18] рекомендуют `x_` и `y_`, но на практике часто встречаются `m_x` и `m_y`. В общем это дело вкуса.

**Вопрос к студентам:** можно ли вызвать метод класса так, чтобы `this` был нулевым указателем?

## 2.7 От программирования в стиле C к программированию в стиле C++

*Most people's C programs should be indented  
six feet downward and covered with dirt*

– Blair P. Houghton

C++ имеет много своих идиом. Многие из них на этом этапе могут быть непонятны (зачем заменять такой хороший malloc на этот странный new?) но переход на активное использование этих идиом лучше начинать уже сейчас, а их подлинная сила откроется позже.

### 2.7.1 От ввода-вывода через функции к потокам

Все мы знаем (а автор этих лекций так и вообще предпочитает) старые добрые функции, которые в C использовались для вывода. Здесь можно их вспомнить в предпоследний раз тихим ласковым словом. Предпоследний, потому что про ввод и выводе ещё предстоит поговорить подробно в (5.5) и там к старым добрым возможностям придётся вернуться ещё раз.

Главной структурой, обобщающей ввод-вывод в C является структура FILE (наследие идеологии Unix, где всё считается файлом, включая такие странные вещи, как консоль или сокет). Поскольку размер этой структуры implementation defined, обычно для совместимости используется FILE\*. Файлы можно создавать, открывать, переоткрывать и закрывать. Количество файлов, которым может оперировать программист задаётся константой FOPEN\_MAX и по стандарту не может быть меньше 8, включая три стандартных файла: `stdin`, `stdout` и `stderr` (первый обычно является read-only и слушает клавиатуру, вторые write-only и направлены на консоль).

Файловый ввод-вывод бывает буферизованным и не буферизованным (традиционно `stdout` буферизован, а `stderr` нет). Буферизованный вывод накапливает определенное количество символов и только потом выводит их в файл, таким образом уменьшая число обращений к физическим устройствам вывода. С другой стороны, при аварийном завершении программы содержимое буфера может испариться, поэтому в таких сценариях лучше использовать небуферизованный вывод.

Буферизация в свою очередь бывает полной и построчной. Скажем, `stdout` буферизован построчно – вывод конца строки сбрасывает буфер на физическое устройство.

Для открываемых вновь файлов, программист может установить режим буферизации через `setvbuf` (C11 7.21.5.6) а также указать собственный буфер любого размера. Очень часто большой буфер в памяти для файла позволяет в разы ускорить работу с ним. Разумеется, в любой момент буфер может быть принудительно сброшен через `fflush`.

Ввод и вывод в файлы в языке C бывает форматированный и неформатированный. Базовые функции неформатированного ввода и вывода это `fputc` и `fgetc`, позволяющие посыпать в файл (читать из файла) один символ. Также к базовым относится функция `ungetc`, позволяющая временно положить назад символ в файл, доступный на чтение. Гарантируется временное помещение назад одного символа (даже для небуферизованных файлов доступных только на чтение, какой-нибудь там клавиатуры, etc). Это позволяет при необходимости “забыть” считанный символ а потом считать его же ещё раз и очень полезно для лексических анализаторов. Чтобы считать таким образом строку символов нужен целый цикл и чтобы избавить от необходимости писать такие циклы, существуют функции `fread` и `fwrite`.

Базовый неформатированный hello world приведен ниже.

```
1 fputs("Hello, world\n", stdout);
```

Форматированный ввод и вывод – самая интересная и спорная часть стандартного вывода. Форматировать можно как вывод в файл (`fprintf`), так и вывод в строку (`sprintf`). С форматированием, прошлый пример будет иметь вид.

```
1 fprintf(stdout, "%s\n", "Hello, world");
```

К сожалению, такой способ вывода небезопасен относительно типов (компилятор не может проверить форматную строку на соответствие типу аргумента). Он подвержен ошибкам, скажем если передать вместо строчки адрес в памяти, то будет выведено содержимое памяти до первого нулевого символа и так далее. Более того – нет возможности расширять набор форматных спецификаторов если захочется устроить вывод для своего класса.

В C++ все синые функции по работе с файлами оставлены, но добавлена новая сущность – поток ввода/вывода (`stream`). Эти потоки не стоит путать с потоками исполнения (`threads`). Поток ввода вывода это

не совсем файл, но он может быть основан на файле (`fstream`) или на строке (`sstream`). Они используются для форматированного ввода или вывода. Стандартные потоки имеют следующее соответствие со стандартными файлами:

| Тип                  | Файл                | Поток                  |
|----------------------|---------------------|------------------------|
| Стандартный ввод     | <code>stdin</code>  | <code>std::cin</code>  |
| Стандартный вывод    | <code>stdout</code> | <code>std::cout</code> |
| Сообщения об ошибках | <code>stderr</code> | <code>std::cerr</code> |
| Логирование          | —                   | <code>std::clog</code> |

Неформатированный вывод осуществляется через методы потоков (о том, что такое метод см. 2.6.4). Например вывод неформатированного Hello, world через `write`

```
1 std::cout.write("Hello, world\n", sizeof("Hello, world\n"));
```

Форматированный вывод осуществляется через несколько странную конструкцию – перегруженный оператор сдвига влево (ввод через сдвиг справо). Подробнее про перегрузку операторов речь пойдёт дальше (см. 3.3). Пока что стоит отметить, что такая перегрузка операторов – очень плохая идея (логический сдвиг не имеет отношения к выводу), но исторически устоявшаяся практика. Для форматирования вывода используются форматные модификаторы.

```
1 std::cout << "Hello, world" << std::endl;
```

Точно так же ввод вывод через потоки может быть буферизован. Потоки ввода-вывода будут рассмотрены подробнее когда речь пойдет про стандартную библиотеку. Пока что может быть неясно (как и со многим в этом разделе) чем они лучше, их подлинная сила вскроется позже.

## 2.7.2 От строк в стиле C к строкам в стиле C++

Язык C++ наследует от C все возможности по работе со строками, завершающимися нулём, которыми так славен последний.

```
1 char astr[] = "hello";
2 char bstr[15];
3 int alen = strlen(astr);
4 assert(alen == 5);
5 strcpy(bstr, astr);
```

```

6  strcat(bstr, ", world!");
7  int res = strcmp(astr, bstr);
8  assert(res < 0);

```

Этот код является одновременно кодом на С и на C++. Увы, для строк в стиле С их длина не является инвариантом. Например:

```
1 int foo (const char *str, size_t len)
```

Нет никакого способа здесь сказать действительно ли `str` это строка и действительно ли её длина это `len` кроме одного: вызвать `strlen` и за линейное время от длины строки убедиться в этом.

Ещё хуже, что при работе с изменяемой строкой легко может возникнуть переполнение буфера и порча памяти (см. 2.4.6 где описаны последствия).

В C++ традиционно для работы со строками используется `std::string`, который для своих целей не сложнее (а местами и проще), чем работа в стиле С. Например, код выше можно переписать следующим образом:

```

1 string astr = "hello";
2 string bstr;
3 bstr.reserve(15);
4 int alen = astr.length();
5 assert(alen == 5);
6 bstr = astr;
7 bstr += ", world!";
8 int res = astr.compare(bstr);
9 assert(res < 0);

```

Синтаксис `astr.length()` и `astr.compare(bstr)` называется вызовом метода и уже рассматривался при разговоре о структурах (см. 2.6.4).

Да, здесь есть некий оверхед, поскольку в случае с C++ строками мы (почти) безальтернативно работаем с кучей, а в случае С строк есть возможность размещать их на стеке. Но иногда это небольшая цена за (сравнительную) безопасность и удобство работы.

Если некая функция требует старых синых строк, класс `string` достаточно гибок и предоставляет возможность притвориться строчкой в старом стиле для старого API

```
1 int res = strcmp(astr.c_str(), bstr.c_str());
```

Разумеется, так следует только читать строки, а не записывать их. Поскольку строка поддерживает длину, запись туда разрешена только через методы строки.

Подобнее о строках разговор пойдёт далее (см. 5.1).

### 2.7.3 От макросов к шаблонным функциям

Ранее, в разделе (2.2.2) рассматривалось то преимущество, которое даёт C++ (и С начиная с C99) позволяя объявлять встраиваемые функции там, где раньше использовались макросы. Но был указан и недостаток – отсутствие возможности передавать аргументы любого типа. На самом деле эта возможность есть. Наивный подход иллюстрируется следующим определением функции:

```
1 template <typename T> inline const T
2 min (const T a, const T b) {
3     return (a < b) ? a : b;
4 }
```

Говорят, что здесь определен шаблон функции. Синтаксис шаблона ясен из определения: после слова `template` в треугольных скобках через запятую перечисляются параметры шаблона. Подробности грамматики изложены в разделе 14 стандарта C++98.

```
1 template <typename T, class U, int x>
```

Здесь слово `typename` означает параметр шаблона, задающий имя типа. По историческим причинам можно использовать `class`, но новый стандарт поощряет использование `typename`. Кроме того вы уже должны видеть в функции `max()` существенный недостаток – параметры, передавающиеся по значению, в случае увесистого Т будут неэффективны, лучше заменить их константными ссылками.

Подробно шаблонные функции рассматриваются в (4.1).

### 2.7.4 От `malloc` и `free` к `new` и `delete`

Для управления динамической памятью в языке С использовались библиотечные функции `malloc` и `free`. Они остались в C++, но их использование во многих случаях не рекомендовано и иногда считается

дурным тоном, поскольку C++ предоставляет гораздо более гибкие возможности с помощью ключевых слов `new`, `delete` и `delete[]`.

```
1 struct S {};
2
3 // Was: t = (struct S*) malloc (sizeof(struct S));
4 struct S *t = new S;
5
6 // Was: free(t);
7 delete t;
8
9 // Was: a = (struct S*) malloc (sizeof(struct S) * 5);
10 struct S *a = new S[5];
11
12 // Was: free(a);
13 delete[] a;
```

Следует оговориться, что для C-подмножества эти возможности по-жалуй менее гибкие и более ограничивающие.

Во-первых нужно помнить о парности `new/delete` и `new a[] / delete [] a`

Попытка выделить память для массива а освободить как для объекта или выделить как для объекта, а освободить как для массива это UB. Попытка выделить через `new`, а освободить через `free` это тоже UB.

Во-вторых нет возможности сделать перераспределение памяти с помощью `realloc`.

Вся сила новых ключевых слов раскроется позже, когда будет рассматриваться создание и уничтожение объектов пользовательских типов (см. 3.1.3). Там же получит обоснование специальная форма для массивов. Пока что можно просто запомнить их и начинать применять.

Впрочем и в C-подмножестве для использования новых ключевых слов можно найти свои примущества. Например, ключевое слово `new` существенно упрощает выделение в памяти многомерных массивов, что было не так то просто в случае языка C. Для C++ можно просто записать:

```
1 typedef int dimensions[3][4];
2
3 dimensions * dim = new dimensions[10];
4 dim[/* 0 to 9 */][/* 0 to 2 */][/* 0 to 3 */] = 42;
```

```
5 delete [] dim;
```

Это не требует убогих трюков когда массив выделяется как одномерный, а потом обслуживается как многомерный – всё прозрачно и типизировано.

**Вопрос к студентам:** каким образом устроен выделенный таким образом трёхмерный массив в памяти?

### 2.7.5 От приведения в стиле С к приведению в стиле C++

Язык С, поскольку он разрабатывался с интенцией отображения один в один на машинные типы, имеет слабую типизацию. Кусок памяти, который хранит int или указатель на int или механическую структуру из чего-нибудь, легко приводится к другому такому же куску памяти.

```
1 int *b, *c;
2 char d;
3 const int *e;
4 int a = (int)(b) + (int)(c); // not in C++!
5 d = (char)a; // not in C++!
6 b = (int *)e;
```

Язык C++ содержит гораздо более развитую систему типов и позволяет определять типы, обладающие состоянием и поведением, о которых пойдёт речь в следующих лекциях. Поэтому, несмотря на то, что формально C++ унаследовал от С способ непосредственного приведения C-style cast, применять его считается крайне плохим тоном. Вместо этого следует применять `static_cast`, `reinterpret_cast` или `const_cast`. Чаще всего вы будете применять `static_cast`. Он записывается так.

```
1 char a;
2 int b = static_cast<int>(a);
```

В принципе это визуально мало чем отличается от C-стиля, рассмотренного выше. Запись чуть более уродлива, зато чуть лучше бросается в глаза в коде программы. Разница в том к чему можно применять `static_cast`, а к чему C-cast. Последний применяется к чему угодно. `static_cast` применяется только к переменным, совместимым по статическим типам: то есть одинаковым по размеру или указывающим на объекты одинакового размера.

```
1 float f = 1.0f;
2 double *d = (double *) &f; // Hmm...
3 double *e = static_cast<double *>(&f); // CE
```

В приведенном примере, компилятор убережет вас от почти неизбежного неопределенного поведения.

Гораздо более редкий `const_cast` нужен для снятия константности и волатильности. С его помощью можно привести `const int *` к `int *`.

```
1 const int *a;
2 int *b = const_cast<int*>(a);
```

Конечно необходимо понимать опасность этих игр: в памяти, выделенной для `const int*`, компилятор размещает данные, изменения которых не ожидает. В тот момент, когда программист принудительно снижает константность и изменяет (пытается изменить) нечто, объявленное ранее константным, он стреляет себе в ногу.

И самый редкий `reinterpret_cast` существует для непереносимых низкоуровневых приведений, которые, тем не менее, иногда нужны.

```
1 int a;
2 int *b = reinterpret_cast<int*>(a);
```

Любое использование `reinterpret_cast` (как и C-cast) компрометирует вашу программу. Но случаи использования `reinterpret_cast` читающему ваш код будут куда проще найти и вычистить.

На самом деле, даже `reinterpret_cast` чуть лучше, чем C-cast. Поэтому что C-cast на самом деле делает следующее (C++14 5.4): пробует сверху вниз перечисленные ниже приведения и подставляет то из них, которое подходит.

- `const_cast`
- `static_cast`
- `static_cast` а потом `const_cast`
- `reinterpret_cast`
- `reinterpret_cast` а потом `const_cast`

Это погружение все ниже и ниже. Хуже того, C-cast вообще способен приводить к ещё не определенным типам.

```
1 struct T; // forward-decl
2 float f = 1.0f;
3 T *t = (T *) &f; // (!)
```

Разумеется, ни одно из трёх рассмотренных преобразований такого не позволяет.

Выучить и использовать три рассмотренных оператора не намного сложнее, чем использовать обычные преобразования в стиле C, но часто они оказываются крайне полезны, упрощая поддержку кода и не давая посадить тяжело обнаружимых ошибок приведения.

## 2.8 Ошибки и исключения

Фундаментальным способом обработки ошибок времени исполнения в языке С являлся любимый препроцессор и макроопределения кодов возврата

```
1 #define E_OK 0
2 #define E_NO_MEM 1
3 #define E_UNEXPECTED 2
```

Разумеется, выше (см. 2.2.1) уже было указано, что стиль C++ здесь это применять перечислимые типы

```
1 enum errors_t { E_OK = 0, E_NO_MEM, E_UNEXPECTED };
```

Подразумевается, что один из этих кодов либо возвращается из функции напрямую, либо возвращается через thread-local facility, например `errno` или `GetLastError`, либо возвращается через `errors_t*` в списке параметров

Ниже приведены три варианта (или если хотите три перегрузки) функции `open_file`

```
1 errors_t open_file (const char *name, FILE **handle);
2 FILE *open_file (const char *name);
3 FILE *open_file (const char *name, errors_t *errcode);
```

В случае чистого С со всеми тремя всё ясно: первая возвращает код ошибки явно, вторая устанавливает `errno`, третья возвращает код ошибки в списке параметров.

В языке C++ к этим вариантам добавляется ещё один: написанная в стиле C++ функция `open_file` скорее всего будет **генерировать исключительную ситуацию** при невозможности его найти или при нехватке прав. То есть в случае C++ второй вариант синтаксиса подозрителен на нечто, генерирующее исключение, а вовсе не устанавливающее `errno`. На самом деле, в случае языка C++ на генерацию исключений в том или ином виде подозрительны все три приведённых выше функции.

К этому нужно быть готовым, поэтому исключения рассматриваются довольно рано.

### 2.8.1 Поддержка исключений в языке

Вообще-то хорошему программисту на С известны некоторые виды исключений. В частности многие системщики хорошо знают:

- Исключительные ситуации уровня аппаратуры (например undefined instruction exception)
- Исключительные ситуации уровня операционной системы (например data page fault)

Исключения в C++ не имеют отношения к этим двум видам исключений и являются механизмом языка, который позволяет нелокальные выходы для сообщения об ошибках (во многом как это происходит при вызове

Поддержка их в целом проста и интуитивна.

Код, обнаруживший исключение, генерирует объект исключения инструкцией `throw <anything>`. Объект исключения может быть любой переменной или даже константой или литералом – язык ничем здесь не ограничивает пользователя (и это уникально, поскольку большинство других языков требуют чтобы объект исключения входил в некую иерархию).

Фрагмент кода выражает своё желание обрабатывать исключение с помощью конструкции `try`. Результатом `throw` является раскрутка стека до тех пор, пока не будет найден подходящий `catch`, которому и передаётся управление.

Ниже приведён некий весьма условный код, написанный в этом стиле.

```
1 FILE *open_file (const char *name) {  
2     if (!__file_exists(name))  
3         throw E_UNEXPECTED;  
4     // .... etc ....
```

При вызове будет зарегистрирован обработчик:

```
1 try {  
2  
3 // a lot of code  
4  
5     open_file("myfile");
```

```
6 // a lot more code
7
8
9 }
10 catch (errors_t x) {
11     switch(x) {
12         case E_UNEXPECTED: // some processing
13     }
14 }
```

Разумеется, работа с исключениями в описанном стиле ужасна. Кидать перечислимый тип – дурной тон, ловить его – практически бесполезно и так далее. Эти примеры нужны только для того, чтобы уяснить базовую идею.

Совсем плохой техникой является перехват всех исключений, который осуществляется через `catch (...)`, но у неопытных программистов такое случается.

```
1 int main () {
2     try {
3         // a lot of code
4     }
5     catch (...) {
6         cerr << "Something bad happened" << endl;
7     }
8 }
```

Исключения являются концептуально сложной техникой и накладывают свои ограничения на разработку. В деталях разговор о них пойдёт ниже (см. 5.2). До этого момента мне остаётся только призвать знать об исключениях, ожидать их иногда, но не использовать их в ваших программах.

## 2.8.2 Отказ от исключений

Есть несколько способов отказаться от исключений в ваших программах пока вы к ним не готовы. Во первых это опция компилятора вроде `--no-exceptions` которая просто убирает возможность их использовать.

Во вторых у вас есть спецификация `throw()`, которую вы можете повесить на функцию, чтобы функция не генерировала исключений.

Например

```
1 FILE *open_file (const char *name) throw();
```

Теперь можно быть уверенным, что эта функция не сгенерирует исключений (а если сгенерирует, то программа будет принудительно завершена).

В третьих, начиная с C++11 полезной аннотацией, которая говорит программисту, что функция или метод не выбрасывает исключений является аннотация через `noexcept`. Казалось бы нет особого отличия её от `throw()` но отличие есть и именно оно делает `noexcept` хорошей идеей.

Что должна делать функция, помеченная как `noexcept` если исключение из неё вылетит? В случае `throw()` стек раскручивается и программа завершается. В случае `noexcept` стек **может быть** раскручивается и программа завершается. В итоге функции помеченные `noexcept` не обязаны генерировать всю поддерживающую исключения обвязку, что позволяет компилятору генерировать гораздо лучший код.

### 2.8.3 Исключения под капотом

Для опытных программистов на С этот раздел показывает как может быть устроено нечто похожее на механизм исключений в этом языке.

Допустим функция `allocate` вызывает `malloc` чтобы выделить n байт, и возвращает указатель, который вернул `malloc`. Если же `malloc` возвращает нулевой указатель, что индицирует нехватку памяти, `allocate` делает нечто похожее на возбуждение исключения `Allocate_Failed` (без размотки стека и вызова деструкторов пока что). Исключение само по себе имеет тип `jmp_buf`, который определён в стандартном хедере `setjmp.h`:

```
1 #include <setjmp.h>
2
3 int Allocation_handled = 0;
4 jmp_buf Allocate_Failed;
```

`Allocation_handled` ноль, пока обработчик не инстанцирован и `allocate` проверяет `Allocation_handled` прежде чем возбуждать исключение:

```
1 void *allocate(unsigned n) {
2     void *newmem = malloc(n);
3     if (newmem)
```

```
4     return newmem;
5     if (Allocation_handled)
6         longjmp(Allocate_Failed, 1);
7     assert(0);
8 }
```

И теперь можно сделать практически `try` и `catch`:

```
1 /* try { */
2 Allocation_handled = 1;
3 if (setjmp(Allocate_Failed)) {
4     goto alloc_except_label;
5 }
6
7 /* ... here some code, calling somewhere allocate() */
8
9 /* } */
10 /* catch (...) { */
11 alloc_except_label:
12 /* ... here processing exception */
13 Allocation_handled = 0;
14 /* } */
```

Разумеется никто в здравом уме не будет делать такое в C++, потому что в C++ раскрутка стека выглядит гораздо сложнее, чем просто дальний переход и требует поддержки компилятора (см. 5.2.2). Но понимание примерной механики того как это может быть устроено в С очень часто благотворно влияет на людей с низкоуровневым мышлением.

Хочется ещё раз призвать не использовать исключения и даже явно запрещать их, если вы не уверены в том, что вы понимаете, что вы делаете. Это сложный и опасный (хотя часто необходимый и мощный) механизм, навык его использования требует наработки, иначе это игра с огнём.

## 2.9 Пространства имён

Каждое объявление в C++ принадлежит некоторому пространству имён (namespace). Это общий термин, описывающий своего рода незримый префикс перед именем.

Допустим некая библиотека предоставляет функцию `open` чтобы открывать картинки по имени файла. Но вы в своём коде тоже хотите функцию `open` чтобы открывать сетевые соединения по имени хоста.

```
1 // piclib.h
2 HANDLE open(const char *);
3
4 // your code
5 #include "piclib.h"
6
7 CONNECTION open(const char *); // ERROR
```

В мире языка С принято манглировать библиотечные имена руками

```
1 HANDLE piclib_open(const char *);
```

Это хрупко, странно выглядит в коде, который не объявляет своей функции `open` и не особо решает проблему. Имя `piclib_open` тоже может кому-нибудь понадобиться.

В мире C++ решением проблемы являются пространства имён

```
1 // piclib.h
2 namespace piclib {
3 HANDLE open(const char *);
4 }
5
6 // your code
7 #include "piclib.h"
8
9 CONNECTION open(const char *); // OK
```

Теперь правильным именем для `open` из библиотеки `piclib` в вашей программе будет `piclib::open`

### 2.9.1 Использование пространств имён и алиасинг

Функции которые не принадлежат никакому пространству имён, принадлежат глобальному. Чтобы специально обратится именно к глобальной функции, используется префикс через два двоеточия.

```
1 void *raw = ::operator new(100);
```

Все стандартные функции принадлежат пространству имён `std`. Ниже приведен пример `Hello world` с явным указанием пространств имён.

```
1 #include <cstdio>
2
3 const char * const helloworld = "Hello, world";
4
5 int main() {
6     std::printf("%s\n", ::helloworld);
7 }
```

Обратите внимание на включение `<cstdio>` вместо привычного заголовочного файла `<stdio.h>`. Все заголовочные файлы к которым вы привыкли в C, сохранены в C++. Но хорошим тоном считается писать унаследованные заголовочные файлы в C++ conforming виде, то есть `<cXXX>` вместо `<XXX.h>`.

Нужно быть осторожным с теми функциями, которые объявлены в стандартной библиотеке языка С как макросы, а не как функции. Разумеется, макросы не могут быть обернуты никаким пространством имен, поэтому правильно писать `assert` а не `std::assert`. Макросы в стандартной библиотеке нужны там, где функциональность может быть отключаема – скажем опция `NDEBUG` позволяет сделать сборку более эффективной, отключив все ассерты в приложении.

Разумеется, гораздо лучше переписать тот же код в новом стиле.

```
1 #include <iostream>
2
3 const char * const helloworld = "Hello, world";
4
5 int main() {
6     std::cout << ::helloworld << std::endl;
7 }
```

Засорять собственными именами, такими как `::helloworld`, глобальное пространство имён это на самом деле крайне плохая идея. Несколько

лучше объявить своё пространство имён, включив туда всё, что специфично именно для вашей программы.

```
1 namespace helloworld {
2     const char * const helloworld = "Hello, world";
3 }
4
5 int main() {
6     std::cout << helloworld << std::endl;
7 }
```

Разумеется, такая педантичность не нужна в маленьких программах и примерах. Однако любая большая программа вынужденно аккуратна с пространствами имён именно до такой степени. В этих лекциях автор обычно избегает таких деталей, но это не должно сбивать с толку: в жизни вам скорее всего придётся писать как-то так.

Если хочется упростить себе жизнь и не писать всюду `std::cout`, можно использовать директиву `using`, которая добавляет в текущее пространство имён все символы из указанного.

```
1 using namespace std;
2
3 int main() {
4     cout << helloworld << endl;
5 }
```

С этой директивой следует быть очень осторожным, потому что она в действительности делает слишком много. Хорошим тоном является вводить в область видимости только нужные имена

```
1 using std::cout;
2 using std::endl;
3
4 int main() {
5     cout << helloworld << endl;
6 }
```

Допустимо объявлять вложенные пространства имён с произвольным количеством уровней вложенности. При помещении в пространство имён функции с большим телом, вполне достаточно поместить в пространство имён явно только объявление (например в заголовочном файле), указав пространство имён при определении (например в файле реализации).

```
1 // foo.h
```

```

2 namespace foo {
3     int bar();
4 }
5
6 // foo.cpp
7 int foo::bar() { /* . . . */ }
```

Пространства имён являются областями видимости (как блоки из фигурных скобок) и подчиняются тем же правилам – если имя указано в охватывающем пространстве имён оно может быть использовано без квалификации. Но в отличии от блоков скобок они могут быть поименованы и тогда переменная или функция с квалификацией может быть использована где угодно

```

1 namespace foo {
2     int y;
3 }
4
5 namespace {
6     int y;
7     void buz(int x) { y = x; } // ok
8 }
9
10 namespace buz {
11     void buz(int x) { foo::y = x; } // ok
12 }
```

Здесь приведён пример неименованного пространства имён (по русски это звучит странно, лучше говорить “анонимное пространство имён”). Объявленные в анонимном пространстве имён функции доступны только в текущем модуле – этим они похожи на функции, объявленные с модификатором `static`

## 2.9.2 Поиск Кёнига

Небольшая проблема, связанная с пространствами имён становится ясна, если рассмотреть как работает стандартный вывод.

```
1 std::cout << "Hello";
```

Здесь на самом деле (см 3.3) написан вызов перегруженного оператора

```
1 operator << (std::cout, "Hello");
```

Но как указать, что сам этот оператор вызывается из пространства имён `std`?

По стандарту это решается так. Когда компилятор видит, например, вызов функции, имя этой функции он будет искать:

- сначала в области видимости вызова и текущем пространстве имён
- далее в пространствах имён аргументов, включая их классы и все базовые классы

Этот трюк называется “поиск Кёнига” по имени человека, который его придумал и ввёл в стандарт C++98. В терминах стандарта он, конечно, так не называется, используется безликий термин ADL (argument-dependent lookup).

```
1 namespace ns {
2     struct S {};
3     void f (S x) {}
4 }
5
6 void g() {
7     ns::S x;
8     f(x); // ok, f is ns::f
9 }
```

Впрочем, если компилятор встречает вызов функции внутри метода некоего класса, то иные члены этого класса и его родительских имеют приоритет над функциями, найденными на основании информации о типах аргументов.

В стандарте C++14 добавлены так называемые встраиваемые пространства имён, которые могут по умолчанию включаться в охватывающее пространство (см. 2.9.3) но без особой нужды (обсуждаемое там же версионирование и т.п.) лучше обходиться рассмотренными здесь средствами.

Поиск Кёнига не срабатывает для имён, которые компилятор считает известными

```
1 typedef int f;
2
```

```

3  namespace N {
4      struct A;
5      int f(A*);
6  }
7
8  int g(N::A *a) {
9      // f is the typedef, not the N::f
10     int i = f(a); // equivalent to int(a)
11     return i;
12 }
```

Без верхнего `typedef` всё сработало бы прекрасно, но в его присутствии, компилятор не будет искать функцию `f`, полагая, что здесь написано преобразование к типу `int`.

Поиск Кёнига также ломается на шаблонных функциях. Особенно это заметно когда не работает вывод типов, например:

```

1  namespace N {
2      struct A;
3      template <typename T> int f(A*);
4  }
5
6  int g(N::A *a) {
7      int i = f<int>(a);
8      return i;
9 }
```

Здесь поломка обусловлена странностями грамматики языка: имя `f` не введено в область видимости как имя шаблонной функции и компилятор считает, что вместо параметризации шаблона записано сравнение на меньше.

Выход: явно ввести имя как имя шаблонной функции:

```

1  namespace N {
2      struct A;
3      template <typename T> int f(A*);
4  }
5
6  template <typename T> void f(int);
```

Несмотря на другие аргументы, такая функция уже будет кандидатом на перегрузку и запустит поиск Кенига по всем остальным местам.

Поиск Кёнига, он же ADL, многие считают весьма страшной аббревиатурой.

```

1 namespace X {
2     bool process(std::string s, int t);
3
4     bool foo() {
5         std::string s;
6         process(s); // may not fail
7     }
8 }
```

Такой код может не быть ошибкой компиляции. Не смотря на то, что это явная опечатка, гипотетически функция `process` может быть найдена через ADL. И если она будет найдена, вас, вероятно, ждут приключений с отладкой.

### 2.9.3 Встраиваемые пространства имён

Встраиваемые пространства имён позволяют по умолчанию включать символы из помеченного таким образом пространства имён в охватывающее. Похоже на то как это происходит с неименованными пространствами имён из (2.9), но, благодаря транзитивности, это их свойство может использоваться для замечательных трюков с версионированием.

Например некая библиотека реализует функциональность работы с сетью. Её изначальный интерфейс мог выглядеть следующим образом.

```

1 namespace Networking
2 {
3     struct TCPSocket;
4     struct UDPSocket;
5 }
```

Но потом появилась необходимость поддержать в программе вторую версию `TCPSocket`.

```

1 namespace Networking
2 {
3     namespace V1
4     {
5         struct TCPSocket;
6     }
```

```
7
8     namespace V2
9     {
10         struct TCPSocket;
11     }
12
13     struct UDPSocket;
14 }
```

Увы, такой подход сломает весь пользовательский код этой библиотеки, так как `Networking::TCPSocket` должен будет везде быть заменен на `Networking::V2::TCPSocket`.

Стандарт C++11 предлагает выход в виде встраиваемого пространства имён, которое неявно включается в охватывающее:

```
1 namespace Networking
2 {
3     namespace V1
4     {
5         struct TCPSocket;
6     }
7
8     inline namespace V2
9     {
10         struct TCPSocket;
11     }
12
13     struct UDPSocket;
14 }
```

В этом случае пользователь сможет продолжать использовать структуру `Networking::TCPSocket` которая по умолчанию будет разрешена в `Networking::V2::TCPSocket`. Также будут работать прямые указания – `Networking::V1::TCPSocket` и `Networking::V2::TCPSocket`.

## 2.10 Структуры и объединения

Очень тонкие и сложные вопросы (причем на удивление коварные) при программировании как на C, так и на C++ связаны с объединениями и структурами.

### 2.10.1 Объединения и каламбуры

Изначально объединения были введены в C для экономии места

```

1 struct VAROBJECT
2 {
3     enum o_t { Int, Double, String } objectType;
4
5     union
6     {
7         int intValue;
8         double dblValue;
9         char *strValue;
10    } value;
11 } object;
```

Здесь используется одна и та же память для всех трёх возможных типов значений – подобно тому как одна комната в отеле может использоваться для трёх разных постояльцев. При этом делается предположение, что постояльцы не пересекаются.

Но очень часто на практике их используют для техники, которая называется “type punning”. Английское слово “rip” обозначает специфическую разновидность юмора (примерно соответствует русскому сленговому слову “прикол”). В русской википедии используется термин “каламбур типа” который кажется мне не слишком удачным. Выглядит эта шутка следующим образом: данные записываются в одно поле объединения, а читаются из другого поля. Таким образом типизация нарушается.

```

1 union a {
2     double d;
3     char pch[10]; /* assume 80-bit double */
4 };
5
6 union a u;
7 u.d = 1.0;
```

```
8 /* use u.pch for per-byte access */
```

Долгое время эта техника была некорректной (в C90 явно заявлено UB в таких случаях) и сильно критиковалась. Но все альтернативы были ещё хуже:

```
1 char pch[10];
2 *(double *)pch = 1.0; /* aliasing violation */
```

Такой код нарушает strict aliasing (2.4.1) и часто ломает те предположения, которые компиляторы считают себя вправе делать относительно типизации указателей и данных. В отличии от этого, type punning ничего не ломает. Поэтому в C++11 и в C11, type punning в явном виде разрешен.

## 2.10.2 Структуры и выравнивание

Способ, которым компилятор размещает данные в памяти ограничен скоростью обращения к ним на соответствующей архитектуре. На большинстве современных архитектур обращение происходит быстрее если данные **выравнены**. Выравнивание каждой переменной часто связано с её размером: переменные типа `int` должны лежать по адресу (в байтах) кратному четырём, типа `short` кратному двум, и так далее.

Это означает, что четыре переменных, объявленных на стеке следующим образом

```
1 char c1;
2 int x;
3 char c2;
4 short y;
```

лядут на условной архитектуре с 32-разрядными указателями примерно так, как показано на (рис. 2.5)

На том же рисунке показано, что простое переупорядочение переменных с учетом выравнивания позволяет сэкономить довольно много места.

Всё то свободное место, которое компилятор оставляет пустым для выравнивания называется **заполнителем (padding)**. В структуре padding обычно идёт между элементами или в самом конце структуры. Например на (рис. 2.5) заполнители отмечены желтым.

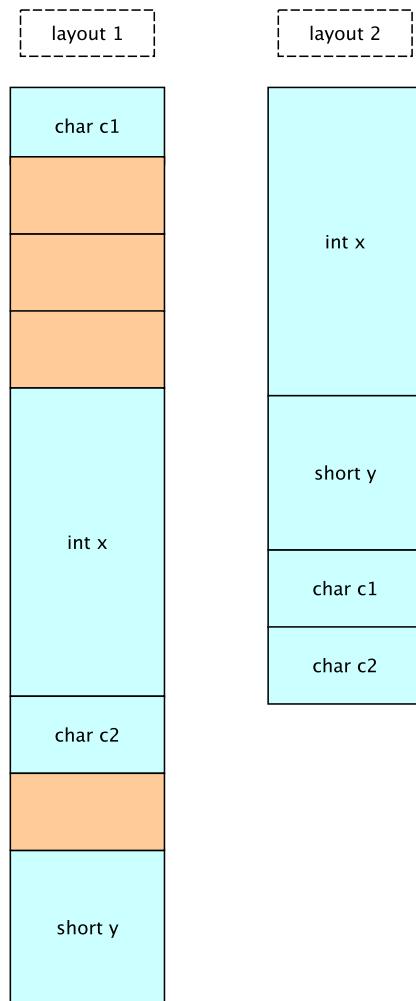


Рис. 2.5: Два способа выравнивания

**Вопрос к студентам:** сколько можно сэкономить на переупаковке вот такой структуры для 32-х битной машины?

```

1 struct foo {
2     char c;
3     char *p;
4     short x;
5 };

```

Даст сокращение с 12 до 8 байт. Это уменьшение расхода памяти на 30% (для массива в котором таких структур сто тысяч такое может быть заметно).

**Вопрос к студентам:** решите ту же задачу для 64-битной а не 32-битной архитектуры.

Структура как целое также имеет своё выравнивание (обычно оно соответствует наибольшему из требований членов структуры).

**Вопрос к студентам:** студент Шустрый предлагает такое переупорядочение структуры с использованием вложенной структуры

```

1 struct foo {
2     struct foo_inner {
3         char *p;
4         short x;
5     } inner;
6     char c;
7 };

```

Сколько займет такая структура в памяти 32-х битной машины?

Ещё одна сложность кроется в работе с битовыми полями:

```

1 struct foo {
2     int flip:1;
3     int nybble:4;
4     int septet:7;
5     short s;
6     char c;
7 };

```

С точки зрения компилятора здесь битовые поля образуют слово, в котором задействованы 12 бит из 16 возможных. В итоге в памяти 32-битной машины, эта структура займет 8 байт. Всё становится гораздо хуже если битовое поле пересекает границу слова.

```

1 struct foo {
2     int bigfield:30;
3     int septet:7;
4     short s;
5     char c;
6 };

```

Такая раскладка заставляет компилятор разместить `septet` в отдельное машинное слово и структура вырастает до 12 байт, из которых 32 бита уходят на семибитное поле.

**Вопрос к студентам:** Когда для какой-либо структуры вызывается

`sizeof`, как вы думаете, он возвращает размер с учётом выравнивания и заполнения или без учёта? И почему.

**Вопрос к студентам:** более хитрый вопрос: а возможно ли вообще написать `sizeof_unaligned` в сколь бы то ни было переносимом виде?

Для C90, 6.2.6.1/6 явно гласит, что при записи любого значения в структуру, биты её паддинга принимают unspecified значение. Этот же пункт был унаследован всеми стандартами С и С++.

**Вопрос к студентам:** рассмотрим структуру

```
1 struct Padded { int x; char y; };
```

В предположении, что `sizeof(char)== 1` и архитектура выравнена на 4 байта, какой размер заполнителя она будет иметь?

**Вопрос к студентам:** допустим структура явно заполнена нулями

```
1 struct Padded p;
2 memset (&p, 0xDA, sizeof(p));
```

Заполнится ли при этом значениями 0xDA её паддинг?

**Вопрос к студентам:** теперь допустим, что некое значение записывается в одно из полей структуры: `p.x = 2`. Можно ли теперь полагаться, что в паддинге осталось значение 0xDA?

### 2.10.3 Прокачка перечислений

Стандартные перечисления хороши, но не без проблем: во-первых они неявно конвертируются в `int`, таким образом создавая возможность использовать их не по назначению. Во-вторых они экспортят имена своих перечислителей в окружающую область видимости, создавая таким образом конфликты имен. В-третьих они не типизированы.

Последнее – самая мрачная проблема:

```
1 enum E_MY_FAVOURITE_FRUITS {
2     E_APPLE      = 0x01,
3     E_WATERMELON = 0x02,
4     E_COCONUT    = 0x04,
5     E_STRAWBERRY = 0x08,
6     E_CHERRY     = 0x10,
7     E_MY_FAVOURITE_FRUITS_FORCE8 = 0xFF
8 };
```

Намерение программиста здесь понятно: хочется, чтобы перечисляемый тип был восьмибитным. Но каким он окажется в реальности? Это зависит от реализации и тут вполне можно ожидать 32 бит.

Все эти проблемы решает появившаяся в C++11 конструкция `enum class`.

```
1 enum class E_MY_FAVOURITE_FRUITS :  
2     unsigned char {  
3         E_APPLE      = 0x01,  
4         E_WATERMELON = 0x02,  
5         E_COCONUT    = 0x04,  
6         E_STRAWBERRY = 0x08,  
7         E_CHERRY     = 0x10,  
8         E_DEVIL_FRUIT = 0x100, // Warning!  
9     };
```

Здесь двоеточие используется для указания нижележащего типа. Таким образом компилятор может выдать предупреждение о несоответствии типов, что может спасти много времени в отладке, а программист может сэкономить на размере.

## 2.11 Автоматический вывод типов

*Whatever we know without inference is mental*

– Bertrand Russel

Вывод типов (type inference) это базовое и очень впечатляющее усовершенствование в стандарте C++11, которое легко понять и легко использовать. Базовая идея просто как правда: вместо того, чтобы указывать компилятору тип, мы даём компилятору свободу вывести допустимый статический тип из контекста определения. С одной стороны это связывает руки: в таких контекстах объявление должно совпадать с определением всегда-всегда. Но с другой стороны, это даёт существенную свободу – свободу не думать о том, какой в этом месте использован тип и почему он там использован. Для многих случаев это очень важно и почти все современные статически типизированные языки дают возможность такого полезного умолчания. Добавление таких возможностей в C++ было неизбежно, а их разумное использование будет для вашей программы благоприятно.

### 2.11.1 Auto и Decltype

Одной из главных сильных черт C++ является его сильная типизированность. Тем не менее часто она утомительна.

```
1 typedef uint_least8_t decl_handler_t;
2 decl_handler_t query_resource (int x);
```

Теперь каждый раз когда нужно вызвать `query_resource`, нужно вспоминать и заводить в программе переменную правильного типа. Или рисковать проблемами, используя для результата `int`.

Но казалось бы, компилятор же знает этот тип. Почему его нельзя просто вывести автоматически? В конце концов это работает для шаблонных функций.

```
1 template <ttypename T>
2 void bar (T x);
3
4 bar (query_resource(0));
```

Здесь будет сгенерирована по шаблону и вызвана функция `bar`, параметризованная типом `decl_handler_t`. Начиная с C++11 то же самое

можно делать для переменных, заводя для них специальный (автоматический) тип.

```
1 auto x = query_resource(0);
```

Теперь тип переменной `x` правильно выводится из реального возвращаемого типа `query_resource`, каким бы он ни был.

Если компилятор может взять тип выражения, логично, что это должен уметь и пользователь. Предыдущий кусок кода может быть переписан так:

```
1 auto i = query_resource(0);
2 decltype(i) j = i++;
```

Здесь на второй строчке объявлено “сделать `j` такого же типа, как `i`”. Это может быть полезно не только для уменьшения кода, но и для облегчения его поддержки:

```
1 /* maybe in future type of a will change */
2 void foo (int a)
3 {
4     /* type of b should be type of a */
5     decltype(a) b;
6     /* do something with b */
7 }
```

Без `decltype` можно банально забыть консистентно исправить типы по всей функции и попасть на неприятные и плохо диагностируемые компилятором ошибки при смешивании знаковых и беззнаковых или типов разного размера.

Иногда `decltype` может быть сопряжен с некоторыми концептуальными проблемами.

```
1 struct Point { int x, y; };
2 Point porig {1, 2};
3 /* ... */
4 const Point &p = porig;
5 decltype(p.x) x; /* int or const int&? */
```

**Тема для обсуждения:** представьте, что вы в комитете по стандартизации. Предложите аргументы в пользу первого и в пользу второго понимания `decltype`, проголосуйте.

Это действительно предмет для спора, так как опытный программист может аргументировать и ту и другую позицию. Автор проводил лекции на эту тему и мнения аудитории обычно разделялись в соотношении примерно 60/40 (причем человеческая интуиция работает **не** в пользу того варианта, который реально принят в стандарте).

Для разрешения этой неоднозначности, стандарт вводит (7.1.6.2) дополнительную форму `decltype` с двумя круглыми скобками:

```
1 decltype(p.x) x; /* int */
2 decltype((p.x)) x; /* const int& */
```

Следует обратить внимание, что, вторая строчка является ошибкой компиляции, так как объявляет неинициализированную ссылку.

На самом деле это различие более тонкое чем просто ещё одни скобки (это была бы слишком грешная магия даже для C++). Речь идёт о разнице между `decltype(id-expr)` и `decltype((expr))`. Первое возвращает тип, с которым было объявлено имя, второе – тип, который могло бы приобрести выражение при его вычислении. Как сделать имя выражением? Обернуть его скобками, очевидно. Это кажется некоторым переусложнением, но это на самом деле единственный разумный выход из ситуации. Увы, `decltype((expr))` имеет неприятную и с первого взгляда необоснованную особенность – если тип под ним это lvalue, то он добавит ссылку (если её не было)

```
1 int x;
2 typedef decltype(x) xval; /* int */
3 typedef decltype((x)) xref; /* int& */
4 typedef decltype((x+1)) xval_again; /* int */
5
6 xval r0;
7 xref r1 = x;
8 xval_again r2;
```

Мотивация для такого поведения станет ясна несколько позже (см. обсуждение в 2.11.2). Но интуитивно это уже ясно – подчеркивается возможность присваивания. Между прочим выражение “могло бы быть” зафиксировано в стандарте и это даёт забавные возможности. Скажем `decltype(10000000)` это “целый тип в котором могло бы поместится 10000000”. Компилятор не обязан выводить наименьший целый тип, но обычно это происходит (в GCC это происходит всегда).

**Вопрос к студентам:** как насчет `decltype((x+y))` если оба `int`?

Вывод типов в C++ работает гораздо более приблизительно, чем их прямая аннотация, поэтому `auto` часто выводит первый попавшийся (на самом деле – наименее общий о чём см. далее) тип, а вот `decltype` всегда старается сохранить даже cv-квалификацию

```
1 const int s = 5;
2 auto s1 = s;
3 decltype(s) s2 = 3;
4 s1 += 1;
5 /* s2 += 1; */
```

Строчка `auto s1 = 3` ничего не изменит.

Разумное применение этих средств позволяет существенно улучшить читаемость вашего кода и сделать гораздо меньше тонких ошибок и опечаток в сложных именах типов. Но тут нужно учитывать, что всегда есть тонкости и волчьи ямы. Я рассказал как ведет себя `auto`, но `auto` ведет себя так не всегда. Если вы конкретизируете тип, то все квалификаторы сохраняются вместе с вашей конкретизацией:

```
1 const int c = 0;
2 auto& rc = c;
3 rc = 44; // error: const qualifier was not removed
```

Также, и это бывает неприятно, `auto` склонно пропускать модификаторы если на верхнем уровне оно встречает то, к чему они относятся:

```
1 int x = 42;
2 const int* p1 = &x;
3 auto p2 = p1; /* p2 is const int* */
4 p2 = &y; /* ok */
5 /* *p2 = 3; */
```

Язык C++ справедливо считает, что константность данных под указателем это слишком важная характеристика типа. В отличии от этого, константность самого указателя может быть сброшена легко:

```
1 const int* const p1 = &x;
2 auto p2 = p1; /* p2 is const int* */
3 p2 = &y; /* ok */
```

Разные правила для `decltype` и `auto` привели в стандарте C++14 к введению идиомы `decltype(auto)` которая позволяет вывести тип из правой части автоматически, но именно так, как его вывел бы `decltype`.

```

1 const int i;
2 auto j = i; /* typeof(j) == int */
3 decltype(auto) k = i; /* typeof(k) == const int */

```

Очень интересный случай это выражение справа от `decltype`, заключенное в круглые скобки.

```

1 int i;
2 auto x4a = (i); /* decltype(x4a) is int */
3 decltype(auto) x4d = (i); /* decltype(x4d) is int& */

```

Таким образом моделируется поведение двух круглых скобок у выражения `decltype` с конкретным `t`.

В качестве вишенки на торте, `auto` на самом деле достаточно мощно, чтобы замещать тип во всех контекстах (пример из стандарта C++14 раздел 5.3.4.2)

```
1 auto x = new auto('a');
```

Здесь выведенным значение будет `char`. Интересно дальше:

```
1 auto x1 = { 'a', 'b' };
```

**Вопрос к студентам:** Что будет выведено здесь?

Интересно, что это – единственное место, где отличается вывод типов шаблонами (см. 2.11) и `auto`. Для шаблонного параметра здесь будет ошибка.

**Вопрос к студентам:** известно, что в случае обычных типов их можно перечислять через запятую в одном определении. Как вы думаете, какой тип будет выведен здесь?

```
1 auto x = 5, *y = &x;
```

## 2.11.2 Расширенный синтаксис функций

Возможности автоматического вывода типов подразумевают построение абстракций с зависимыми типами. Пусть необходим статический шаблонный контракт на любой тип `T`, поддерживающий функцию `T::makeobject`, возвращающую некий свой тип. Нельзя (по крайней мере в стандарте C++11) просто взять и написать:

```
1 template <typename T> auto /* Error! */
```

```

2 makeAndProcessObject (const T& builder)
3 {
4     auto val = builder.makeObject();
5     /* do stuff with val */
6     return val;
7 }
```

Потому что компилятор в точке объявления функции не обладает информацией о типе, который вернет `T::makeObject`. Забегая вперед – иногда обладает, скажем в C++14 тут все хорошо. Точно так же не сработает вот такой выход:

```

1 template <typename T>
2 decltype(builder.makeObject()) /* Error again! */
3 makeAndProcessObject (const T& builder)
```

Потому что `builder` не может быть использован до точки своего объявления (которой является список аргументов функции). Конечно правильного прошаренного пацана это не остановит. Он использует тот факт, что значение под `decltype` не вычисляется, и сделает тонко:

```

1 template <typename T>
2 decltype(((T*)0)->makeObject()) /* painfull but works */
3 makeAndProcessObject (const T& builder)
```

Но нельзя требовать от программистов такое всерьез, всегда и везде. Комитет по стандартизации решил эту проблему изящно, предложив расширенный синтаксис для обобщённых функций, возвращающих зависимые типы:

```

1 template <typename T> auto
2 makeAndProcessObject (const T& builder) -> decltype (builder.
3     makeObject())
4 {
5     auto val = builder.makeObject();
6     /* do stuff with val */
7     return val;
8 }
```

Внутри скобок `decltype` в данном случае вычисление выражения (в том числе вызов функции) не происходит – происходит только вывод типа.

Выбор правильного результата для гетерогенных функций это большее дело, например:

```

1 template<typename A, typename B>
2 auto add(A const& a, B const& b) -> decltype(a + b)
3 {
4     return a + b;
5 }
```

Этот подход будет работать во всех случаях, когда входные типы допускают сложение.

Конечно, здесь есть возможные ошибки и засады:

```

1 template <typename T, typename S>
2 auto min(T x, S y) -> decltype(x < y ? x : y)
3 {
4     return x < y ? x : y;
5 }
```

Казалось бы все при всем, но так писать нельзя, поскольку `decltype` вокруг выражения работает как `decltype(expr)`, а значит результат может быть выведен как ссылка, что чревато. Так как же все таки правильно написать type-generic minimum? Эта проблема найдет свое решение позже (см. 3.5.5).

**Вопрос к студентам:** Представим, что `T` и `S` это простые типы без квалификаций, указателей ссылок и прочего добра. Просто `int`, `double` и всё такое. В каких случаях `decltype` здесь выведет ссылку, а в каких нет?

Зато теперь можно понять почему же такое поведение `decltype(expr)` было выбрано комитетом по стандартизации. Рассмотрим упрощенную задачу – допустим речь идет о выводе типа для доступа к элементу массива:

```

1 template <typename T>
2 auto array_access(T& array, size_t pos) -> decltype(array[pos])
3 {
4     return array[pos];
5 }
```

С текущим подходом можно использовать такую обертку прозрачно как если бы это действительно был доступ к элементу массива:

```

1 std::vector<int> vect = {42, 43, 44};
2 int* p = &vect[0];
3
```

```
4 array_access(vect, 2) = 45;
5 array_access(p, 2) = 46;
```

В противном случае пришлось бы идти на разнообразные хаки.

Позвольте еще маленькую вишненку на этот тортик:

```
1 auto main() -> int
```

Теперь является легитимной формой функции `main` и если вы хотите показать насколько вы круче остальных людей в этом мире, то вы теперь знаете, что делать.

Впрочем, это так, отвлечение. Последний стандарт привнес ещё больше радости.

### 2.11.3 Обобщения вывода типов функциями

В некоторых простых случаях компилятору действительно не составляет проблем вывести тип функции:

```
1 auto isquare (int x) -> decltype (x) {
2     return x*x;
3 }
```

Здесь указание `decltype` выглядит просто излишним и C++14 разрешает его убрать:

```
1 auto isquare (int x) { return x*x; }
```

Для таких простых вариантов все хорошо, но как быть с рекурсией? Здесь возникает проблема: тип должен быть выведен до того, как рекурсивный вызов произошел:

```
1 auto sum_to (int i) {
2     if (i < 2)
3         return i; // return type deduced as int
4     else
5         return sum_to (i-1) + i; // ok to call it now
6 }
7
8 cout << sum_to (10) << endl;
```

Но если переставить возвраты в вышеприведенном коде, он не будет скомпилирован.

```

1 auto bad_sum_to (int i) {
2     if (i > 2)
3         return bad_sum_to (i-1) + i;
4     else
5         return i;
6 }
```

Впрочем, GCC возвращает вполне человечное описание ошибки

```
error: use of ‘auto bad_sum_to(int)’ before deduction of ‘auto’
      return bad_sum_to (i-1) + i;
```

С 2014 года также нет проблем с тем, чтобы шаблонные методы выводили свой тип непосредственно из других методов того же класса без явного `this` (пример взят из стандарта C++14, 5.1.1.3)

```

1 struct A {
2     char g();
3     template <typename T> auto f(T t) {
4         return t + /* this-> */ g();
5     }
6 };
```

Одно опасное заблуждение связано с идеей использовать `auto` в списке параметров функции (не лямбды, для лямбд всё хорошо, см. 4.7.1)

Люди пишут нечто вроде следующего кода (обратите внимание на `auto` в списке параметров), предполагая, что компилятор выведет нечто из аргумента по умолчанию.

```
1 auto foo (auto x = 1) { return x; }
```

Во-первых это вообще не C++. Во-вторых, это скомпилируется только благодаря популярному расширению в GCC которое ввели, когда казалось, что в 2017 году такой синтаксис примут в стандарт (нет, не приняли). В третьих, даже как расширение это будет работать вот так

```
1 template <typename T> auto foo (T x = 1) { return x; }
```

То есть никакого вывода до точк вызова. Здесь `auto` в списке типов просто заменяет явный шаблонный аргумент.

Это крайне неочевидный синтаксис, он был плох для включения в стандарт, он плох как расширение, не надо так делать.

### 2.11.4 Точный вывод и прозрачная оболочка

Увы, как уже было сказано `auto` режет типы

```
1 auto Example(int const& i) { return i; }
```

Здесь возвращаемый тип `int`. Конечно, в конкретном коде несложно вернуть квалификацию типа:

```
1 auto const& Example(int const& i) { return i; }
```

Но что делать в обобщенном коде?

В обобщенном коде для точного вывода возвращаемого типа может быть использован `decltype(auto)` например так:

```
1 template <typename Fun, typename Arg>
2 decltype(auto) transparent(Fun fun, Arg arg) {
3     return fun(arg);
4 }
```

Теперь тип возвращаемого значения будет проброшен точно. Ниже будет показано как написать совершенно прозрачную обертку: пробросив не только возвращаемое значение но и произвольные аргументы (см. 3.5.6).

### 2.11.5 Decaying и минимальные общие типы

Decaying уже был рассмотрен, при рассмотрении деградации массивов в указатели (3), но если в старом стандарте это была built-in особенность для одной конкретной пары типов, то новый стандарт предлагает интересные варианты обобщения этого понятия на любые типы. Простой случай:

```
1 int foo (const int &s) { return s + 2; }
```

Здесь в выражении `s + 2`, `s` ведёт себя так, как будто его тип `int`. Тогда можно сказать, что `const int &` деградирует к `int` в том же смысле, в каком массив деградирует к указателю, etc. Новый стандарт позволяет вручную “деградировать” тип:

```
1 const int &i;
2 std::decay<decltype(i)>::type j; // int j
3 auto k = i;                      // int k = i;
```

Автоматический вывод также осуществляет деградацию типов. Поэтому можно считать `decay + decltype` способом вывести тот тип, который вывело бы `auto`.

На механизме `decay` неявно построен механизм `common_type`, позволяющий вывести минимальный общий тип:

```
1 template <typename T, typename S> void foo(T lhs, S rhs) {
2     std::common_type<decltype(lhs), decltype(rhs)>::type k;
3     // .... etc ....
4 }
```

В принципе минимальные общие типы не так уже и нужны

```
1 template <typename T, typename S>
2 void foo(T lhs, S rhs) {
3     auto prod = lhs * rhs;
4     // ...
5 }
```

Устроит не менее качественную деградацию. Мало того, такое расширение GCC давно известное и во многих других компиляторах как `typeof`, позволяло делать это и в старом стандарте. В новом же можно использовать `decltype`.

```
1 typedef typeof(lhs * rhs) product_type;
2 typedef decltype(lhs * rhs) product_type;
```

В качестве полезного примера, можно привести смешанную арифметику для числового класса:

```
1 template <typename T> struct Number { T n; };
2
3 template <typename T, class U>
4 Number<typename std::common_type<T, U>::type>
5 operator+(const Number<T>& lhs,
6             const Number<U>& rhs) {
7     return {lhs.n + rhs.n};
8 }
9
10 int main() {
11     Number<int> i1 = {1}, i2 = {2};
12     Number<double> d1 = {2.3}, d2 = {3.5};
13     std::cout << "i1i2: " << (i1 + i2).n
```

```
14      << "\ni1d2: " << (i1 + d2).n
15      << "\nd1i2: " << (d1 + i2).n
16      << "\nd1d2: " << (d1 + d2).n
17      << '\n';
18 }
```

**Домашняя наработка:** проанализировать вывод этой программы и то, почему он ведет себя именно так.

## 2.12 Домашняя наработка по базовому C++

### Контрольные вопросы

1. Запишите волатильный указатель на массив целых чисел
2. Приведите пример когда использование `typedef` серьёзно улучшает читаемость кода
3. В каких случаях программист на C++ вынужден пользоваться макросами?
4. В чём преимущество констант над перечислениями?
5. Возможна ли онлайн-подстановка нестатической функции, определенной в другой единице трансляции?
6. В каких случаях вызов функции вместо подстановки идентичного макроса ускоряет работу кода?
7. Может ли объявление структуры располагаться внутри определения функции?
8. Может ли время жизни объекта заканчиваться раньше достижения границ его области видимости?
9. Можно ли использовать ссылку на неполный тип в определении функции?
10. Может ли вызов функции быть lvalue?
11. Можно ли вычислить побитовое или двух указателей?
12. Возможно ли иметь в программе массив из ссылок?
13. Возможна ли ссылка на массив?
14. Всегда ли можно получить указатель на нечто, на что существует ссылка?
15. В каких случаях нельзя заменить `static cast` на `const cast`?
16. Может ли функция быть перегружена по cv-квалификатору аргумента?

17. Можно ли расширить пространство имен, определяемое структурой?
18. Есть ли разница в выводе типа auto из правой части или правой части в дополнительных скобках?
19. Можно ли использовать auto в полях структур?
20. Какой тип будет выведен для пустой строки?
21. Какие преимущества имеет новый синтаксис объявления функций?
22. Какой тип является минимально общим между ссылкой на int и ссылкой на float?

### Задания

1. Данна структура данных, воплощающая простой односвязный список

```
1 typedef struct list_tag
2 {
3     void *data;
4     struct list_tag *next;
5 } list_t, *list_p;
```

У последнего элемента **next = 0**

Необходимо написать на языке C++ функцию, берущую на вход указатель на голову списка и переворачивающую список в памяти, так, что первый элемент становится последним, второй предпоследним и так далее.

2. Данна программа на языке С с комментариями вида /\* comment \*/

Необходимо написать на языке C++ программу, выкидывающую все комментарии из текста данной.

3. Данна структура данных, соответствующая n-арному дереву

```
1 typedef struct tree_tag
2 {
3     void *data;
4     struct tree_tag *top;
5     struct tree_tag **child;
6 } tree_t, *tree_p;
```

Реализуйте на языке C++ функцию, берущую на вход пару указателей на произвольные элементы дерева, и подсчитывающую расстояние между ними (минимальный путь в дереве). Как вы будете тестировать эту функцию?

4. Реализуйте на языке C++ алгоритм из (2.1.4) и протестируйте на нескольких сотнях сгенерированных определений
5. Реализуйте на языке C++ транспонирование двумерной матрицы
6. Реализуйте на языке C++ операцию получения двумерной матрицы обратной данной
7. Реализуйте на языке C++ операцию умножения вектора на матрицу
8. Реализуйте на языке C++ калькулятор, делающий вычисления в обратной польской нотации. Например  $1\ 2\ 3\ 4\ +\ *\ + =$  должно выдавать 25 в качестве результата.
9. Известны год, месяц и день рождения человека. Реализуйте на языке C++ программу, определяющую его возраст в днях на текущую дату.
10. Реализуйте на языке C++ программу, подсчитывающую количество лет в 20-м веке у которых первым днём было воскресенье
11. На вход дана строка из  $N*N$  символов. Реализуйте на языке C++ функцию, выделяющую в памяти двумерную матрицу  $N*N$  и заполняющую её последовательными данными из входной строки.
12. На вход даны завершающаяся нулём строка `haystack` и завершающаяся нулём строка `needle`. Реализуйте на языке C++ функцию, определяющую, является ли `needle` подстрокой `haystack`
13. В условиях предыдущей задачи необходимо доработать функцию, чтобы она выкидывала из `haystack` все вхождения `needle` и возвращала измененную строку.
14. Напишите на C++ программу, которая будет возвращать номер в последовательности Фибоначчи первого числа имеющего 1000 разрядов в десятичном представлении.

## Глава 3

# Объектно-ориентированное счастье

*I made up the term 'object-oriented',  
and I can tell you I didn't have C++ in mind*

*– Alan Kay, OOPSLA '97*

Как известно, самой целью создания C++ Бьёрном Строструпом было добавление объектно ориентированных возможностей к C, поэтому изначально язык назывался “C с классами”. Известно также, что Строструп вдохновлялся языком Simula, но сейчас уже нельзя оценить насколько это была удачная идея, поскольку этот язык канул в лету (зато известна вынесенная в эпиграф цитата одного из гуру ООП). Так или иначе, но C++ действительно обладает уникальной среди современных объектно ориентированных языков моделью объявления и инстанцирования классов. Многим нравится богатство и гибкость её возможностей, многие в ужасе отползают. Эта глава посвящена особенностям ООП в C++ и является центральной темой первого семестра обучения.

### 3.1 Инкапсуляция и игра в мячик

*I find OOP philosophically unsound.  
 It claims that everything is an object.  
 Even if it is true it is not very interesting.*

– Alex Stepanov

Пусть стоит задача разработать тип данных, который будет использован для моделирования полёта материальной точки в двумерном мире (высота, длина). Мяч может лететь свободно, для чего у него вызывается метод `fly(double t)` или его можно толкнуть, придав ему определённую скорость под определённым углом (после чего например опять отправить в полёт и так далее). Можно написать определение структуры, вроде приведенного ниже.

```

1 struct ball {
2     double t;
3     double vx;
4     double vy;
5     double x;
6     double y;
7     void push(double v, double alpha);
8     void fly(double time);
9 }

```

Представляет ли эта структура данных хорошую абстракцию мяча? Нет, не представляет. Каждый пользователь вашего “мяча” может произвольно менять его координаты. Это означает, что в плохо отлаженной программе симуляции, этот мяч сможет свободно “телеортироваться”, а это явно не то, чего ждут от законченной модели.

```

1 ball foo() {
2     ball a_ball;
3     a_ball.vx = 5.0; /* sad, but ok */
4     a_ball.push(5.0, 0.75); /* ok */
5     a_ball.x = -1.0; /* hmm... still ok */
6     return a_ball;
7 }

```

Хуже того, время может быть свободно переставлено вперёд или назад. Но пусть даже никто не ошибся и всё написано правильно. А потом...

возникла необходимость “запустить” этот мяч в многопользовательской среде. Всё пропало – для того, чтобы вставить синхронизацию, переписывать придётся каждый участок кода где ссылались на эти поля.

### 3.1.1 Конкретные классы

Для разграничения состояния модели от её поведения и более гибкого управления поведением, в C++ были введены классы. Простые классы, без использования полиморфизма и без иерархий наследования, называются “конкретными классами”. Конкретные классы – мощный и полезный инструмент для поддержания консистентности абстракции. Можно переписать модель мяча, создав его класс

```

1 struct ball_t {
2     private:
3         double m_x;
4         double m_y;
5         double m_vx;
6         double m_vy;
7         double m_t;
8     public:
9         void push(double a_v, double a_alpha);
10        void fly(double a_time);
11    };

```

Модификатор `public` означает, что любой пользователь типа `ball_t` имеет доступ к этим методам или данным.

Модификатор `private` означает, что доступ к соответствующим методам и данным имеют только методы этого класса.

```

1 void ball_t::push(double a_v, double a_alpha) {
2     assert(a_v > 0);
3     m_vx += a_v * cosf(a_alpha); /* ok */
4     m_vy += a_v * sinf(a_alpha);
5 }
6
7 ball_t bar() {
8     ball_t ball;
9     ball.m_vx = 5.0; // FAIL
10    ball.push(5.0, 0.75); // OK
11    return ball;

```

12 };

Хорошим тоном считается закрывать данные, составляющие состояние объекта и открывать функции, составляющие его поведение.

**Вопрос к студентам:** вам приносят код, в котором автор написал класс, открывающий к полю как чтение так и запись:

```
1 struct colored_ball_t {
2     private:
3         int color_;
4         // all other fileds ....
5     public:
6         int get_color() { return color_; }
7         void set_color(int color) { color_ = color; }
8         // all other methods ....
9     };
```

Его коллега предлагает убрать обе функции и вынести поле `color_` в открытую часть, так как оно всё равно не инкапсулировано. Что вы на это скажете?

**Вопрос к студентам:** вы сидите в комитете и вам приносят пример:

```
1 struct Bar {
2     private:
3         int foo (int);
4     public:
5         int foo (char);
6     };
```

7

```
8 Bar b;
9 b.foo(1);
```

Как по вашему: должна быть ошибка или вызов `public` функции?

### 3.1.2 POD и NPOD типы

В языке С со структурами всё было довольно просто.

Из-за скрытия состояния и наличия нетривиального поведения, в языке C++ существует ряд категорий типов: типы со специальным созданием, со специальным поведением при копировании, с неизвестным или зависящим от реализации размером и так далее.

Стандарт определяет следующие термины (C++14, 9 раздел) для обозначения категорий типов, объекты которых могут быть созданы.

- **Тривиально копируемый** – тип, непрерывный в памяти и не обладающий специальным поведением при копировании. Для таких типов работает `memcpy` и `memmove`
- **Тривиальный** – тривиально копируемый тип без специального поведения при создании.
- **Со стандартным расположением полей** – тип, без зависящих от реализации полей (например без членов-ссылок, без таблиц виртуальных методов и так далее). Такие типы удобны для передачи в модули, написанные на других языках программирования, например на C или Java – их всегда можно десериализовать независимым от языка и компилятора образом.
- **POD** – от plain old data – тривиально копируемый тип со стандартным расположением полей.

Все типы, не являющиеся POD называются NPOD (от not plain old data).

По мере введении новых техник ООП, будет рассмотрено каким становится тип в зависимости от объявления.

Типы с закрытой частью остаются тривиально копируемыми.

### 3.1.3 Инициализация и уничтожение

Отсутствие доступа к состоянию означает, что при создании объекта он должен уметь сам установить своё состояние, а при уничтожении – освободить свои ресурсы. Для этого в класс вводятся конструктор и деструктор – специальные функции, вызывающиеся при создании и уничтожении объекта. Например у класса мяча, конструктор может устанавливать начальное положение, деструктор же может быть тривиальным.

```

1 ball_t(double a_x = 0.0, double a_y = 0.0):
2     m_x(a_x), m_y(a_y), m_vx(0.0), m_vy(0.0), m_t(0.0) {}
3
4 ~ball_t() {}

```

Обратите внимание на список инициализации у конструктора в этом примере кода. Можно, конечно, написать инициализацию в теле конструктора, но использование списков инициализации является лучшей идеей. По умолчанию, удовлетворяющий стандарту языка C++ компилятор генерирует конструктор (на самом деле несколько конструкторов) и деструктор по умолчанию. Это кажется удобным, но в 3.2.1 последует обсуждение того, чем это может быть чревато. Конструктор по умолчанию вызывает конструкторы всех членов класса, у которых они есть, деструктор – их деструкторы.

Здесь становится ясна упомянутая ранее (см. 2.7.4) сила новых ключевых слов `new` и `delete`. При необходимости создать мячик на куче, обычный `malloc` выделит память, но не вызовет конструктор. Единственный способ вызвать конструктор это использовать `new`. Простейший способ получить мячик на куче:

```
1 ball_t *pball = new ball_t(1.0, 2.0);
2 // use it here
3 delete pball;
```

Соответственно двадцать мячиков инициализированных по умолчанию получаются через вызов специального `new` с квадратными скобками. Он не только вызывает конструкторы всех этих объектов, но ещё и записывает специальные структуры в памяти, чтобы сообщить следующему вызову `delete[]` сколько деструкторов вызывать.

```
1 ball_t *pballs = new ball_t[20];
2 // use it here
3 delete [] pballs;
```

В базовом ООП, выделение памяти и вызов конструктора неразрывны. Когда речь пойдёт о переопределении операторов, будет рассмотрена возможность тонкого управления выделением и освобождением памяти (см. 3.4) в том числе так называемые “размещающие” формы `new`. Тогда окажется, что это не всегда так. Но пока что можно считать, что это так.

Очень часто масса конструкторов в классе делает одно и то же или почти одно и то же. В этих случаях, начиная с C++11, конструкторы можно **делегировать** – то есть вызвать конструктор того же класса из списка инициализации.

```
1 class X {
2     int a;
3 public:
```

```

4   X(int x) : a(x) { /* ... some logic ... */ }
5   X(double x) : X(static_cast<int>(x)) { }
6 };

```

Во многих случаях, инициализация списком утомительна. Для того, чтобы сократить записи списков инициализации, была придумана инициализация в теле класса.

```

1 class A {
2     int a = 7;
3     int b = 5;
4 public:
5     A() {} // sets a = 7, b = 5
6     A(int a_val) : a(a_val) {} // sets b = 5
7 };

```

Она так же доступна начиная с C++11 и её очень хорошо использовать чтобы точно не оставить ни одного поля в неинициализированном состоянии.

Неожиданно сложным вопросом является вопрос о том, что делать если в конструкторе произошла ошибка. Короткий ответ: использовать исключения. Подробный ответ будет рассмотрен далее (см. 5.2.1). Пока что предполагается, что все конструкторы в этом разделе очень просты и ошибок, требующих нелокальной обработки, в них просто не будет.

### 3.1.4 Неявное преобразование типов и explicit

Конструкторы также используются для задания неявного преобразования типа. Неявные преобразования есть и в C, там они называются type promotions, некоторые из них (действуют и в C++) сведены в таблицу ниже (здесь anytype это любой встроенный тип, совместимый по операции но не перечисленный выше):

```

1 anytype 'op' long double => long double 'op' long double
2 anytype 'op' double => double 'op' double
3 anytype 'op' float => float 'op' float
4 anytype 'op' unsigned long long => unsigned long long 'op'
    unsigned long long
5 anytype 'op' long long => long long 'op' long long
6 anytype 'op' unsigned long => unsigned long 'op' unsigned long
7 anytype 'op' long => long 'op' long

```

```

8 anytype 'op' unsigned int => unsigned int 'op' unsigned int
9 anytype 'op' int => int 'op' int

```

Но в C++ неявные преобразования также пробуются компилятором для пользовательских типов. Определение в пользовательском типе конструктора, который может быть истрактован как конструктор с одним аргументом (считая аргументы по умолчанию частично подставленными всюду кроме первого аргумента), считается определением неявного преобразование из аргумента конструктора к этому типу. Это может иметь неприятные последствия:

```

1 int emulateBall(ball_t b);
2
3 int foo() {
4     emulateBall(1.0); // ??? but legal
5 }

```

Верный способ определить конструктор, чтобы явно заявить компилятору, что он не поддерживает неявного преобразования это определить его с ключевым словом `explicit`

```

1 explicit ball_t(double a_x = 0.0, double a_y = 0.0):
2     m_x(a_x), m_y(a_y), m_vx(0.0), m_vy(0.0), m_t(0.0) {}

```

Это важное решение при проектировании и его надо принимать осознанно. Лепить `explicit` куда ни попадя – дурной тон (скажем это ключевое слово возможно но совершенно не нужно на конструкторе более чем с одним аргументом и без аргументов по умолчанию). Но иногда он очень нужен.

### 3.1.5 Value-инициализация и Default-инициализация

Тривиальные и нетривиальные типы требуют разной обработки во время выделения памяти. Когда пользователь пишет

```

1 int *t = new int; /* t uninitialized */
2 int *pt = new int [10]; /* pt[.] uninitialized */

```

Он не ожидает, что будет вызвано десять конструкторов для целых чисел. Вместо этого ожидаемое поведение это десять неинициализированных объектов.

```

1 struct SomeTriv

```

```

1 {
2     int x;
3 }
4
5
6 SomeTriv *px = new SomeTriv; /* x uninitialized */

```

Аналогично никакого конструктора не будет вызвано и в этом случае. Но немного изменим структуру:

```

1 struct SomeNTriv
2 {
3     int x;
4     ~SomeNTriv() {};
5 };
6
7 SomeNTriv *px = new SomeNTriv; /* x initialized with 0 */

```

Поскольку в этой структуре есть деструктор, она теперь не тривиальна и для неё будет сгенерирован конструктор по умолчанию. Для того, чтобы сделать у тривиальных типов такое же поведение, разработчики языка предусмотрели указание пустых скобок для value-initialization:

```

1 struct SomeTriv
2 {
3     int x;
4 };
5
6 SomeTriv *px = new SomeTriv(); /* x initialized with 0 */
7 int *t = new int(); /* t initialized with 0 */
8 int *pt = new int[10](); /* 10 ints are initialized with 0 */

```

Это создаёт некоторую запутанность, зато даёт некий аналог `malloc` для всех тривиальных типов.

### 3.1.6 Селекторы

Отражает ли созданная до сих пор абстракция физический мяч? Всё ещё нет. Мяч в физическом мире обычно виден пользователю, у которого есть возможность считать его координаты. То есть нужны некоторые методы, которые будут, сохраняя состояние мяча, давать возможность прочитать его. Такие методы традиционно называются селекторами.

```

1 double get_x() const { return m_x; }
2 double get_y() const { return m_y; }
```

Обратите внимание на `const` в их объявлении. Внутри объявленного таким образом метода изменить любое поле класса это ошибка компиляции. Исключение составляют поля, объявленные с ключевым словом `mutable`, но им не стоит злоупотреблять при проектировании.

**Домашняя наработка:** найти нетривиальный случай, когда ключевое слово `mutable` полезно и оправдано

Хорошой считается привычка делать селектором любой метод, который теоретически может быть селектором и по логике не должен менять внутреннего состояния объекта. Это позволяет дополнительные оптимизации в компиляторе и аннотирует метод важной информацией для дальнейшего переиспользования.

### 3.1.7 Статические члены в классе

Данные в классе, образующие его состояние, называются атрибутами объектов класса, потому что каждый объект имеет свою копию такого поля (такое поле `x` является только на самом деле `this->x`). Именно поэтому методы в структуре или классе берут первым аргументом неявный указатель на тот объект, для которого они вызваны (см. 2.6.4).

Но иногда необходимо иметь общий атрибут для всех объектов класса. В этом случае используется ключевое слово `static`, которое в контексте класса означает, что методы и данные не зависят от `this` и являются общими для всех объектов класса.

```

1 class ball {
2     // attributes
3     int x, y;
4     // static field
5     static int ball_count;
6     public:
7     ball() : x(0), y(0) { ball_count += 1; }
8
9     // static method
10    static int get_count () {
11        return ball_count;
12    }
13 }
```

```

14
15 int ball::ball_count = 0;

```

Здесь каждый конструктор при создании очередного мяча увеличивает счетчик мячей и статический метод отдаёт этот счетчик. Можно обратить внимание, что он не `const`. Это не опечатка, более того, он и не может быть константным: поскольку у такого метода нет объекта класса, он не может изменять состояние ни одного объекта.

```

1 static int get_count () {
2     x += 1; // oops? We have no this.
3 }

```

Таким образом это отношение работает в одну сторону: все объекты класса имеют доступ к его статическим методам и данным но не наоборот (без специальных ухищрений).

По сути статические методы не сильно отличаются от глобальных функций. Единственное отличие – у статических методов класса есть доступ к закрытому статическому состоянию класса. Точно так же и статические члены не сильно отличаются от глобальных переменных. Нужно обратить особое внимание на инициализацию статического члена вне класса. Это вполне естественно: такие вещи нельзя делать в конструкторе, потому что статический член должен быть доступен даже если ни одного объекта ещё не создано.

Дополнительное определение где-то вне класса требуется даже если инициализации не нужно.

```

1 struct S {
2     static int x;
3 };
4
5 int S::x;

```

Это связано с тем, что выделение памяти в конструкторе (как для объектов классов и их полей) не происходит, но память должна быть где-то физически выделена. Надо быть осторожнее с ODR – плохая идея делать это в заголовочнике.

Иключение оставлено для статических констант, объявляемых внутри классов. Они допускают инициализацию по месту объявления.

### 3.1.8 Указатели на методы класса

*If you can't program in a language with ugly warts,  
maybe C++ isn't the language you should be programming in*

– John Calb

Указатели на статические методы класса по сути не отличаются от указателей на функции. Указатели на нестатические члены класса сложнее, потому что вызов по такому указателю должен содержать объект класса для которого производится вызов. Можно рассмотреть простой класс (даже без закрытой части) о котором известен один его метод.

```
1 struct MyClass {
2     int DoIt(float a, int b) const;
3     // ...
4 };
```

Тип “указатель на константный метод” тогда может быть легко записан (звёздочка после двойного двоеточия может смузить, но это часть синтаксиса).

```
1 typedef int (MyClass::*constif_t)(float, int) const;
2 using constif_t = int (MyClass::*)(float, int) const;
```

Теперь вызов состоит в получении указателя на метод и связывании его с объектом.

```
1 constif_t ptr = &MyClass::DoIt;
2 MyClass c;
3 (c.*ptr)(1.0, 1);
```

Интересный синтаксис возможен если есть указатель на такой класс и указатель на его метод

```
1 MyClass *d = &c;
2 (d->*ptr)(2.0, 2);
```

Это печально знаменитый оператор `->*`, который осваивают немногие, но который получает вторую жизнь в контексте умных указателей.

### 3.1.9 Объявления и определения классов

Это важный момент, перекликающийся с затронутой в (2.3.1) темой объявлений и определений. Объявление класса как неполного типа выглядит так:

```
1 class ball_t;
```

С этого момента тип `ball_t` можно использовать по правилам, прописанным в стандарте для неполных типов. Определение класса это объявление всех его методов и полей.

Но внутри определения класса, каждое объявление поля, статического поля или метода это его объявление. Определением нестатического члена считается конструктор класса (поэтому если членом класса является ссылка она должна быть инициализирована в списке инициализации конструктора). Определение метода может как встречаться внутри класса, так и быть вынесено вне его. Определение статического объекта всегда должно быть вне класса.

**Домашняя наработка:** свести весь код игры в мячик воедино, добиться работоспособности кода

## 3.2 Классы для управления ресурсами

*C++ is my favorite garbage collected language  
because it generates so little garbage*

– Bjarne Stroustrup

В идеальном мире, программа производит чистые вычисления над неограниченным входным потоком и записывает результаты в неограниченный выходной поток. Реальный мир вносит корректиды: хорошо написанная программа всегда работает в условиях недостаточности машинных ресурсов и должна сама заботиться о том, чтобы рационально управлять запросами и освобождением ресурсов. До сих пор основным видом ресурсов с которыми вы сталкивались была динамическая память. Хорошо написанная программа рационально выделяет себе нужное количество динамической памяти и вовремя её освобождает. Возможность тонкого ручного управления ресурсами – важная особенность языков С и С++

**Вопрос к студентам:** какие ещё вы знаете ресурсы.

Что общего у всех этих ресурсов? Как правило – наличие парных команд для запроса и освобождения. Скажем это:

- `new` и `delete` для динамической памяти,
- `fopen` и `fclose` для файлов,
- `mysql_real_connect` и `mysql_close` для запросов MySQL C API,
- `pthread_mutex_init` и `pthread_mutex_destroy` для работы с мьютексами в POSIX.

В общем случае, можно говорить о паре функций: некоей функции `query` запрашивающей ресурс и функции `release` освобождающей его. Если ресурс имеет семантику общего владения, добавляется ещё функция `addref`, добавляющая ресурсу пользователя, но в общем это уже экзотика. Ограничимся пока моделью из функций `query` и `release`. Рассмотрим типичный код на С-подмножестве С++, работающий с выделением динамической памяти и дополнительно неким ещё ресурсом `res_t`

```
1 int foo (int n) {  
2     int *a = new int[n];  
3     res_t other = query();
```

```

4 // .... some code ....
5 if (condition) {
6     delete [] a;
7     release (other);
8     return FAILURE;
9 }
10 // .... some code ....
11 delete [] a;
12 release (other);
13 return SUCCESS;
14 }
```

В этом коде очевидна проблема с проектированием: код освобождения дублируется многократно, часто в непредсказуемых местах. Сначала рассмотрим какие выходы обычно используются в legacy code на языке С или подобных ему.

Как ни странно, лучший выход, официально применяемый в ядре Linux, это использование в таких случаях `goto`.

<https://www.kernel.org/doc/Documentation/CodingStyle>

Смотреть седьмую часть документа или просто поиском.

```

1 int *a = new int[n];
2 res_t other = query ();
3 int errcode = SUCCESS;
4
5 // .... some code ....
6 if (condition) {
7     errcode = FAILURE;
8     goto cleanup;
9 }
10 // .... some code ....
11
12 cleanup:
13 delete [] a;
14 release (other);
15 return errcode;
```

Вообще при упоминании `goto` люди нехорошо напрягаются. Использование таких конструкций в языках высокого уровня некрасиво и отчасти может приводить к проблемам, описанным ещё Дейкстрой.

Вторым вариантом является трюк с использованием `do-while`, который позволяет организовать `goto` без явного `goto`:

```

1 int *a = new int[n];
2 res_t other = query ();
3 int errcode = SUCCESS;
4
5 do {
6     // .... some code ....
7     if (condition) {
8         errcode = FAILURE;
9         break;
10    }
11    // .... some code ....
12 } while (0);
13
14 delete [] a;
15 release (other);
16 return errcode;
```

Этот поучительный код организует цикл нулевой длины, эксплуатируя возможности языка по принудительному выходу из таких циклов. Он выглядит криво и косо, но, бывает, встречается. Таких мест не надо пугаться – люди просто боялись `goto` и этот страх породил чудовищ.

Второй выход известен как вложенная функция – логика отдаётся в особую функцию, управление ресурсами которой происходит извне. Такой подход также можно часто встретить в открытом коде, например в GCC. Для приведенного примера это будет:

```

1 int foo1 (int *a, int n, res_t other) {
2     // .... some code ....
3     if (condition)
4         return FAILURE;
5     // .... some code ....
6     return SUCCESS;
7 }
8
9 int foo (int n) {
10    int *a = new int[n];
11    res_t other = query ();
12
13    int errcode = foo1 (a, n, other);
```

```
14
15     delete [] a;
16     release (other);
17     return errcode;
18 }
```

Но этот метод имеет свои недостатки – он как минимум создаёт лишний вызов функции и запутывает код. “Всего лишь ещё одна” обёрточная функция прошённая себе десять раз это плюс десять уровней косвенности при отладке.

Третий выход известен как oksofar trick, от английского “Ok so far” == “Всё пока что [идёт] хорошо”. Он предлагает рассматривать функцию как последовательность состояний:

```
1 int *a = new int[n];
2 res_t other = query ();
3 int errcode = SUCCESS;
4 int oksofar = 1;
5
6 // .... some code ....
7
8 if (oksofar) {
9     // .... some code ....
10    if (condition) {
11        errcode = FAILURE1;
12        oksofar = 0;
13    }
14 }
15
16 if (oksofar) {
17     // .... some code ....
18 }
19
20 delete [] a;
21 release (other);
22 return errcode;
```

Этот ужас также можно наблюдать в реальных проектах.

### 3.2.1 Идиома RAII

Создатели языка C++ прекрасно знали все эти трюки языка С и применяли их не по разу. Но зная их, они их не любили.

Поэтому в современном C++ при программировании очень часто используют важную идиому **Resource Acquisition Is Initialization** сокращённо RAII (выделение ресурса это инициализация), создавая для каждого ресурса обёрточный объект, в котором ресурс будет захвачен в конструкторе и освобождён в деструкторе. Эта же идиома расширяется на так называемые “умные указатели” о которых разговор пойдёт позднее.

Проще всего

```
1 struct Buffer {
2     Buffer (int n) : m_a (new int[n]) {};
3     ~Buffer () {delete [] m_a;};
4     int *ptr() const {return m_a;};
5 private:
6     int *m_a;
7 };
8
9 struct Resource {
10     Resource () : m_res(query()) {};
11     ~Resource () {release(m_res)};
12     const res_t &res() const {return m_res;};
13 private:
14     res_t m_res;
15 };
16
17 int foo (int n) {
18     Buffer a(n);
19     Resource other;
20
21     // .... some code ....
22     if (condition)
23         return FAILURE;
24     // .... some code ....
25
26     return SUCCESS;
27 }
```

Обратите внимание на то как элегантно выделение и освобождение ресурсов теперь происходят только в те моменты когда они должны происходить. Но в разработанных выше классах `Buffer` и `Resource` есть одна общая важная уязвимость, которая будет пояснена далее.

### 3.2.2 Переопределение копирования

Но сначала немного теории. Интересный объект рассмотрения – пустой, сферический класс в вакууме.

```
1 class Empty {};
```

Так ли он пуст, как это кажется на первый взгляд? Совершенно очевидно, что даже объект такого совершенно пустого класса в программе на C++ может быть создан, создан по образцу, скопирован и разрушен. Всю эту связку, если её не предоставляет программист, предоставляет компилятор C++. То есть написанное определение эквивалентно следующему:

```
1 /* There is nothing empty in C++ */
2 class Empty {
3 public:
4     Empty() { /* default implementation */ }
5     Empty(const Empty &rhs) { /* default implementation */ }
6     ~Empty() { /* default implementation */ }
7     Empty& operator=(const Empty &rhs) { /* default
        implementation */ }
8     // .... and maybe something else ....
9 };
```

Я люблю использовать для этого идиому “смотреть через волшебные очки”. Вроде бы программист смотрит на пустой класс, а на самом деле видит там шесть методов. Пока что видно четыре, но волшебные очки на то и волшебные, что нуждаются в тонкой настройке, которая будет завершена вскоре (см. 3.5.4) но не сейчас.

Конструктор и деструктор должны быть к этому моменту уже понятны. Но оказывается, что C++ генерирует ещё две функции специального вида – копирующий конструктор, отвечающий за создание объекта по образцу такого же и оператор присваивания, который будет вызван при присваивании объекта в выражениях вида `a=b`.

```
1 Empty e1;
```

```

2 Empty e2(e1); // copy constructor called
3 e1 = e2; // assignment called

```

По умолчанию сгенерированные компилятором конструктор копирования и оператор присваивания просто побитово копируют код аргумента в целевой объект. Разумеется, это не пройдёт если в классе есть член-ссылка, тогда вы должны писать оператор присваивания самостоятель-но.

**Домашняя наработка:** объяснить почему присваивание по умолча-нию не годится при членах класса, являющихся ссылками.

Иногда такое умолчательное поведение приводит к мрачным про-блемам, тяжёлым в отладке. Предположим, вы написали некий класс, управляемый внутренним буфером, который выделяется в конструкто-ре и освобождается в деструкторе.

```

1 class CBuffer {
2     char *m_buffer;
3     int m_size;
4 public:
5     CBuffer(int size = 10) { m_size = size; m_buffer = new char[
6         size]; }
7     ~CBuffer() { delete[] m_buffer; }
8     char &get(int x) { assert(x >= 0 && x < m_size); return
9         m_buffer[x]; }
10 };

```

А потом кто-то по незнанию создал буфер по его образцу внутри какой-то функции.

```

1 void wrong() {
2     CBuffer b1;
3     CBuffer b2 = b1;
4 } /* Segfault here */

```

Что при этом произойдёт? Выделенный вами буфер доступен по ука-зателю. Указатель будет побитово скопирован. Это значит что теперь на буфер есть два указателя. При выходе за границы блока оба будут освобождены. Это крайне неприятная ошибка двойного освобождения, которое является UB.

Говорят, что по умолчанию C++ реализует поверхностное копирова-ние (shallow copy). Копирование с выделением нового буфера и копиро-ванием в него содержимого старого это глубокое копирование (deep copy)

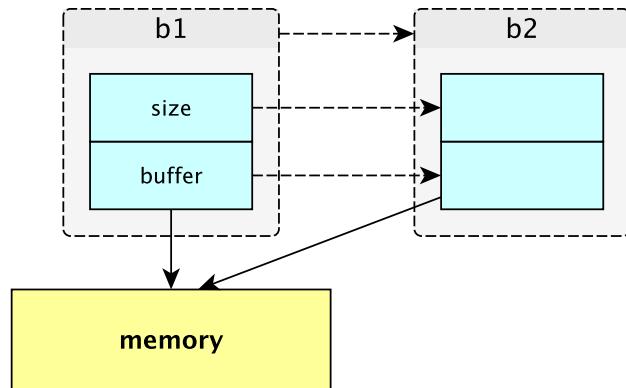


Рис. 3.1: Ошибка двойного освобождения

которое пользователь всегда должен реализовать самостоятельно. Ниже показано переопределение копирующего конструктора, о переопределении операторов и в частности оператора присваивания см. (3.3)

```

1 class CCopyableBuffer {
2 public:
3     CCopyableBuffer(int size) :
4         m_size (size) m_buffer (new char[m_size]) {
5     }
6     ~CCopyableBuffer() {
7         delete[] m_buffer;
8     }
9     CCopyableBuffer(const CCopyableBuffer& rhs) {
10         m_size = rhs.m_size;
11         m_buffer = new char[m_size];
12         memcpy(m_buffer, rhs.m_buffer, m_size);
13     }
14     char &get(int x) {
15         assert(x >= 0 && x < m_size);
16         return m_buffer[x];
17     }
18 private:
19     char *m_buffer;
20     int m_size;
21 };

```

Так всё будет работать. Но иногда копирование не нужно и его проще запретить, объявив собственные конструктор копирования и оператор

присваивания в закрытой части класса.

```

1 class NCBuffer {
2 public:
3     NCBuffer(int size) :
4         m_size (size) m_buffer (new char[size]) {
5     }
6     ~NCBuffer() {
7         delete[] m_buffer;
8     }
9     char &get(int x) {
10        assert(x >= 0 && x < m_size);
11        return m_buffer[x];
12    }
13 private:
14     char *m_buffer;
15     int m_size;
16     NCBuffer(const NCBuffer& rhs);
17     NCBuffer& operator= (const NCBuffer& rhs);
18 };

```

Теперь компилятор не сгенерирует за вас неправильные варианты и копирование просто не будет скомпилировано.

Начиная с 2011 года, существует более цивилизованный способ запретить любой генерируемый по умолчанию метод или явно сгенерировать его – ключевые слова `default` и `delete`.

```

1 class NCBuffer {
2 public:
3 // .....
4     NCBuffer(const NCBuffer& rhs) = default;
5     NCBuffer& operator= (const NCBuffer& rhs) = delete;
6 // .....
7 };

```

Человечное определение класса наподобие `Empty`, которое не требует от программиста “волшебных очков”.

```

1 struct Empty {
2     Empty() = default;
3     ~Empty() = default;
4     Empty(const Empty &rhs) = delete;
5     Empty& operator=(const Empty &rhs) = delete;

```

```
6 };
```

**Домашняя наработка:** поясните или опровергните мысль: “именно из-за проблемы двойного освобождения, в C++ нет и не может быть простого аналога realloc”

Конечно писать RAII-обертку на каждый ресурс это очень накладно (в англоязычной литературе любят использовать в таких случаях термин boilerplate code). Поэтому профессионалы пишут одну обертку на все типы ресурсов – так называемые умные указатели, которые будут рассмотрены в (5.3).

### 3.2.3 Оптимизации возвращаемого значения

RVO или оптимизация возвращаемого значения, прописана в стандарт C++ (например 12.3.2.15 для C++98) и она гласит, если коротко, что компилятор имеет право не вызывать копирующие конструкторы если он может статически доказать, что объекты эквивалентны.

Пример:

```
1 struct foo {
2     foo () { cout << "foo::foo()" << endl; }
3     foo (const foo&) { cout << "foo::foo( const foo& )" << endl;
4         }
5     ~foo () { cout << "foo::~foo()" << endl; }
6 };
7
8 foo bar() {
9     foo local_foo;
10    return local_foo;
11 }
12
13 int main() {
14     foo f = bar();
15     return 0;
16 }
```

В случае если GCC компилирует этот код без RVO (опция `-fno-elide-constructors`), вывод на экран выглядит как:

```
1 foo::foo()
2 foo::foo( const foo& )
```

```
3 foo::~foo()
4 foo::foo( const foo& )
5 foo::~foo()
6 foo::~foo()
```

Последовательность очевидна: создаётся локальный `local_foo`, создаётся его копия – возвращаемое значение, уничтожается `local_foo`, возвращаемое значение копируется в `f`, уничтожается возвращаемое значение, уничтожается `f`.

В случае же обычной компиляции с RVO, вывод выглядит гораздо проще:

```
1 foo::foo()
2 foo::~foo()
```

Эта экономия разрешена стандартом и поэтому программист не должен закладываться на то, что его конструктор копирования всегда будет вызван в контексте копирования.

### 3.3 Перегрузка операторов

*I left out operator overloading  
because I had seen too many people abuse it in C++*

– James Gosling

Язык C++ позволяет переопределять для пользовательских типов почти все операторы, которые применимы для встроенных типов, включая арифметические, логические, а также некоторые специального вида операторы. В прошлом разделе уже встречалось упоминание переопределения присваивания. Рассмотрим класс, абстрагирующий матрицу  $N \times M$  для  $N > 1, M > 1$ .

**Вопрос к студентам:** написать у доски то, что они уже знают: конструктор, деструктор, копирование, геттер/сеттер

Допустим, хотелось бы написать сложение матриц.

```
1 Matrix& Matrix::addeq(const Matrix &rhs)
```

Использование очевидно.

```
1 Matrix m, n;
2 // ...
3 m.addeq(n);
```

Но это не так красиво, как если бы допускалась более математическая запись с инфиксным сложением.

```
1 m += n;
```

И C++, как вскоре станет ясно, даёт возможность переопределить операторы, такие как `+`, `=` и многие другие, чтобы добиться такой непосредственности в выражении мыслей.

Главная проблема здесь в том, что вы действительно вольны написать любой код в определении оператора сложения. Ваше сложение не обязано удовлетворять даже каким-то базовым инвариантам сложения:

```
1 assert (a + b == b + a); /* ORLY? */
```

Вместо этого ваша операция сложения может осуществлять вычитание, лезть в базу данных или форматировать диск. Почти всегда, когда вы читаете код на C++, вы обязаны иметь в виду, что не можете быть

уверены что значит операция + этим утром. Ночью кто-нибудь мог вкомутиить в неё чудовищные изменения – из лучших побуждений, разумеется.

Всё усугубляется тем, что общий список операторов, которые можно переопределить, крайне внушающ. Его всегда можно посмотреть в стандарте, но кроме основных арифметических и логических операций, определение которых довольно таки прямолинейно, перегрузке могут подвергаться совершенно экзотические вещи – сравнение, присваивание, приведение к типу (любому, включая встроенные), обращение по указателю и даже выделение памяти с помощью new и её освобождение. Давайте побеседуем об нескольких специальных и проблематичных случаях переопределённых операторов.

### 3.3.1 Операторы, формирующие цепочки

Первый особый случай, это операторы, которые могут быть записаны цепочками (обычно – правоассоциативными). Скажем, пользователь оператора =, скорее всего ожидает корректной работы строчки:

```
1 x = y = z = w;
```

распространяющей, значение w справа налево. Рассмотренный ранее (3.2.2) буфер может и должен быть расширен определение оператора равенства, который для того, чтобы такая цепочка могла быть сформирована, должен возвращать ссылку на текущий объект

```
1 CCopyableBuffer& operator=(const CCopyableBuffer& rhs){
2     if (&rhs == this) return *this;
3     delete [] m_buffer;
4     m_size = rhs.m_size; m_buffer = new char[m_size];
5     memcpy(m_buffer, rhs.m_buffer, m_size);
6     return *this;
7 }
```

Особое внимание следует обратить на проверку `if (&rhs == this)` в начале. Вторым способом написать эту проверку будет: `if (rhs == *this)`

**Домашняя наработка:** почему второй способ может привести к некорректному и неожиданному поведению?

Большинство цепочечных операций: +=, \*= и так далее, являются прямыми модификаторами состояния класса и должны быть определены в

классе, чтобы у них был `this` для возврата.

### 3.3.2 Симметричные бинарные операторы

Можно рассмотреть ещё один пример – разработку класса, реализующего абстракцию комплексных чисел.

Для того, чтобы абстракция комплексных чисел была завершённой, должна быть возможность неявного преобразования `double` в `complex`, ведь по сути число `2.0` это `2.0 + 0.0 * i`. Эта возможность заложена в конструкторе класса (вспоминаем, что конструктор с одним аргументом это неявное преобразование типа).

```
1 Complex a = Complex(1.0, 3.1), b;
2 b = a + 2.0; /* ok */
```

Но здесь таится и опасность, потому что сложение перестаёт быть коммутативным и простая запись, вида:

```
1 Complex a = Complex(1.0, 3.1), b;
2 b = 2.0 + a; /* error! */
```

Выдаст ошибку компиляции. Ошибка связана с тем, что неявные преобразования применяются только к параметрам, перечисленным в списке параметров. Поэтому `a.operator+(2.0)` преобразует `double` к `Complex`, но для `(2.0).operator+(a)` неявный параметр `this` – указатель на объект для которого вызывается метод, не подвергается, согласно стандарту, никаким неявным преобразованиям.

Правильный метод: определить операторы сложения и умножения вне класса как отдельные функции. Очень часто бинарные операции проще всего определить в терминах цепочечных операций, определенных в классе.

```
1 class Complex {
2     double re, im;
3 public:
4     Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
5
6     double get_real() const { return re; }
7     double get_imag() const { return im; }
8     double operator+=(const Complex &rhs);
9 };
```

```

10
11 Complex operator+ (const Complex &lhs, const Complex &y) {
12     Complex tmp = lhs;
13     tmp += rhs; // ----- call to operator +=
14     return tmp;
15 }
```

Точно так же необходимо поступать со всеми функциями, которые по определению должны быть коммутативными. Тогда коммутативность будет сохранена.

На самом деле, можно сделать и иначе через friend функции, но это предпочтительный метод. Он кроме всего прочего позволяет естественным образом сохранить когерентность + и += или в общем случае – цепочечной и соответствующей ей бинарной операции.

### 3.3.3 Инкремент и декремент

Язык C++ получил своё название от унарной операции `x++`, которая увеличивает значение переменной в данном случае `x`, и возвращает старое значение. Такая операция называется постинкрементом. Известен также прединкремент `++x`, который увеличивает свой аргумент и возвращает увеличенное значение.

```

1 Complex& Complex::operator++ () {
2     re += 1.0;
3     return *this;
4 }
```

Такая запись возвращает прединкремент, он совсем простой. Постинкремент в терминах прединкремента может быть определен вот так:

```

1 Complex operator++ (int unused) {
2     Complex result = *this;
3     ++(*this);
4     return result;
5 }
```

Очевидно, что прединкремент в общем случае не требует копирования, которое, как уже выяснилось, может быть очень дорогим. Поэтому очень часто программисты на C++ используют именно прединкремент в циклах:

```
1 for (Complex t = 1.0; t < 5.0; ++t) { .... }
```

вместо

```
1 for (Complex t = 1.0; t < 5.0; t++) { .... }
```

Обычно замеры показывают что такая запись не несет какого-то особых выигрыша. Она просто показывает, что человек знает внутреннюю механику языка.

### 3.3.4 Вызов и индексация

Эта перегрузка позволяет объекту быть вызванным как функция (с круглыми скобками и передачей аргументов).

```
1 class LessThan
2 {
3     public:
4         explicit LessThan(int x = 0) : m_x(x) {}
5         bool operator()(int y) const { return y < m_x; }
6     private:
7         int m_x;
8 }
```

Определённый таким образом класс, позволяет иметь сразу несколько функции-подобных объектов (также они называются **функции-члены**), выраждающих предикаты “меньше x” для разных x.

```
1 LessThan ltthree(3), ltfour(4), ltfive(5);
2
3 assert(ltthree(2));
4 assert(!ltthree(4));
```

И вызывать их по необходимости.

Аналогично возможно перегрузить операцию взятия индекса.

```
1 struct TwoArray {
2     int y, int z;
3     int& operator[](int x) {
4         if (x == 0) return y;
5         if (x == 1) return z;
6         // error processing
7     }
```

```
8 }
```

При использовании такого класса, синтаксис мало чем отличается от массивов

```
1 TwoArray t;
2 t[0] = 1;
3 t[1] = 2;
4 assert (3 == t[0] + t[1]);
```

Если не знать, невозможно догадаться, что реально никакого массива никто не выделяет.

Переопределение квадратных скобок часто используется контейнерами стандартной библиотеки, в частности векторами и деками, см. (5.4.2).

### 3.3.5 Разыменование и его варианты

Если есть классы, которые ведут себя как функции и массивы, логично предположить, что возможны классы, которые ведут себя как указатели. Для примера, можно рассмотреть так называемые умные указатели, о которых речь пойдет чуть дальше (см. 5.3). Пусть есть несложная структура

```
1 struct T {
2     int x = 5;
3     int foo () { return x; }
4 };
```

Можно изготовить класс, который будет вести себя почти как указатель на эту структуру

```
1 class Tptr {
2     T *t_;
3 public:
4     Tptr(T *rhs) : t_(rhs) {};
5     T operator*() const { return *t_; }
6     T* operator->() const { return t_; }
7 };
```

В этом классе перегружены операторы обычного разыменования и разыменования с доступом к полю (часто для краткости его называют “стрелкой”). Использование несложно.

```

1 T t;
2 Tptr pt(&t);
3 pt->x = 6;
4 (*pt).foo();

```

Одной из важных особенностей перегрузки оператора стрелки является *drill-down behaviour* – зарывающееся поведение: результат стрелки используется для разрешения имени, но если этого имени нет, то вместо него снова может быть выбран вызов стрелки.

```

1 struct client { int a; };
2
3 struct proxy {
4     client *target;
5     client *operator->() const { return target; }
6 };
7
8 struct proxy2 {
9     proxy *target;
10    proxy &operator->() const { return * target; }
11 };
12
13 void f() {
14     client x = { 42 };
15     proxy y = { & x };
16     proxy2 z = { & y };
17
18     // prints 42 42 42
19     printf ("%d %d %d\n", x.a, y->a, z->a);
20 }

```

**Домашняя наработка:** в самой структуре T теперь можно перегрузить `operator&` который мог бы возвращать `Tptr`, и таким образом, переписать код использования ещё проще. Попробуйте заставить это заработать.

В любом языке есть элементы бессмысленной джигитовки. В частности в C++ есть вызов по указателю на функцию:

```

1 struct foo {
2     int bar() { printf ("bar\n"); }
3 };
4

```

```
5 template <typename T>
6 void Apply (T* t, int (T::*bar)() ) {
7     (t ->* bar) ();
8 }
9
10 int main () {
11     int (foo::*pbar)() = &foo::bar;
12     foo f;
13
14     Apply (&f, pbar);
15
16     return 0;
17 }
```

Скобки необходимы в `Apply` поскольку приоритет `->*` меньше, чем приоритет вызова функции.

Этот оператор тоже можно перегрузить. В перегрузке он самый неинтересный из всех операторов разыменования и ведет себя просто как произвольная бинарная операция. На самом деле он даже не обязан быть нестатическим!

Но помните: если вы обнаруживаете, что вы перегрузили **это**, значит у вас точно что-то пошло не так.

### 3.3.6 Приведение типов

Самая простая категория доступных для определения операторов это операторы приведения типов.

Для двух любых типов X и Y может быть четыре оператора приведения:

- Приведение Y к X, объявленное внутри X
- Приведение X к Y, объявленное внутри Y
- Приведение X к Y, объявленное внутри X
- Приведение Y к X, объявленное внутри Y

При этом каждое приведение может быть явным или неявным.

Первые два вида это обычные конструкторы. Пусть в качестве Y служит тип `int` для простого примера, указатель на целое для чуть более сложного примера и (указатель на) массив из трёх целых для совсем сложного случая. Да, конечно, можно сказать, что нельзя придумать такой класс, который требовал бы неявного приведения к указателю на массив из трёх целых и обратно. Но я бы с этим поспорил.

```

1 struct X {
2     X(int) {} // converting ctor int -> X
3     X(int*) {} // converting ctor int* -> X
4     X(int(*)[3]) {} // converting ctor int(*)[3] -> X
5
6     // implicit conversion X -> int
7     operator int() const { return 7; }
8
9     // explicit conversion X -> int*
10    explicit operator int*() const { return nullptr; }
11
12    // Error: array operator not allowed in conversion-type-id
13    // operator int(*)[3]() const { return nullptr; }
14
15    // OK if done through typedef
16    using arr_t = int[3];
17    operator arr_t*() const { return nullptr; }
18
19    // Error: conversion to array not allowed in any case
20    // operator arr_t () const;
21 };

```

Использование довольно просто и поучительно

```

1 X x(1); // init ctor conversion
2
3 X x1 = 2; // again init ctor
4
5 X x2 = static_cast<int>(3); // again init ctor
6
7 int n = static_cast<int>(x); // OK, explicit
8 int m = x; // OK, implicit
9
10 int *p = static_cast<int*>(x); // OK, explicit
11 int *q = x; // Fail: no implicit one

```

```
12  
13 int (*pa)[3] = x; // OK
```

Интересно, что, являясь самым простым случаем, приведение является также самым разнообразным, потому что из любого типа можно делать неявное приведение в любые другие, в том числе пользовательские, что создаёт поле для бесконечных игр

```
1 struct S {};  
2  
3 struct T {  
4     operator S();  
5 };
```

**Вопрос к студентам:** что вы думаете о неявном преобразовании к неполным типам?

## 3.4 Выделение и освобождение динамической памяти

В отличии от многих современных языков программирования, таких как Java, Python или Ruby, где работа с памятью скрыта от программиста, в языке C++ память это ресурс и тонкая настройка памяти это важный навык.

Ключевую роль в тонкой настройке памяти играет переопределение операторов выделения и освобождения памяти. Эта возможность – нечто новое даже по сравнению с таким низкоуровневым языком как С, где, тем не менее, вся память почти безальтернативно приходит из единственной функции `malloc`.

В C++ напротив, `new` и `delete` это операторы и в этом смысле они похожи на такие операторы как `+` или `-`, то есть являются чем-то, что можно переопределить.

В частности навык переопределения этих операторов понадобиться когда речь пойдёт о безопасности исключений (см. 5.2) и о пользовательских аллокаторах (см. 5.10).

### 3.4.1 Глобальные операторы

Самым близким аналогом функций `malloc` и `free` являются глобальные операторы `::operator new` и `::operator delete`. Они используются каждый раз когда в классе не переопределены `new` и `delete` либо для встроенных типов.

Очень часто они напрямую реализованы через механизмы языка С, так что начинающему программисту кажется, что отличий нет. Тем не менее,, отличий довольно много.

Функции `malloc` и `free` отличаются тем, что:

1. Память аллоцируется из кучи (“Heap”)
2. Возвращаемое значение имеет тип `void*`
3. При неудаче `malloc` возвращает `NULL`
4. Требуют спецификации точного размера в байтах

5. Выделение массива требует ручного подсчёта места на массив
6. Перевыделение памяти просто (нет вызова копирующих конструкторов)
7. Никогда не вызовут из себя `new` или `delete`
8. Нет способа автоматически побить запрашиваемую память на более мелкие участки при нехватке памяти на большой линейный кусок
9. Не могут быть легально переопределены ни в каком случае

В отличие от них, `::operator new` и `::operator delete` имеют следующие свойства:

1. Память аллоцируется из свободного хранилища (“Free Store”). Совпадает ли оно с кучей (“Heap”) упомянутой выше или это два разных места – зависит от реализации. Обычно в Unix совпадает, так как используется системный вызов `sbrk`, но есть варианты.
2. Возвращаемое значение – типизированный указатель
3. Обычно выбрасывает исключения при неудаче (см. ниже про специальные формы)
4. Размер памяти вычисляется компилятором исходя из данного типа
5. Есть версия специально для массивов
6. Перевыделение памяти сложно (из-за вызова конструкторов) и нет его стандартной поддержки (аналога `realloc`)
7. Вызовут ли они из себя `malloc` или `free` зависит от реализации
8. Поддерживают установку аллокатора для случая исчерпания памяти через `set_new_handler`
9. Могут быть легально переопределены для пользовательских типов
10. Используют конструкторы для инициализации объектов и деструкторы для освобождения их ресурсов

### 3.4.2 Стандартные и специальные формы new

Рассмотрим типичное выделение памяти в программе на C++:

```
1 class Widget {};
2 ...
3 Widget *w = new Widget;
```

Предположим, оператор `new` не переопределён для класса `Widget`

Что здесь происходит? Неожиданно много всего.

- Сначала, по форме оператора `new`, компилятор определяет какой именно `new` имеется в виду. Существуют три нормальные формы `new`:

- `new` в куче с возбуждением исключений при исчерпании памяти. Именно он использован в рассматриваемом примере.
- `new` в куче с возвратом нуля при исчерпании памяти
- `placement new` (размещающий `new`) (с размещением в заданном месте). Именно он и был использован в листинге выше. Его синтаксис похож на выделяющий `new` из этого примера, но буфер, в котором размещается сконструированный объект, не выделяется, а передаётся как параметр в скобках.

Кроме нормальных существуют ещё и специальные формы `new`, когда пользователь некоего типа переопределяет размещающий `new` этого типа передавая ему не область памяти, а нечто иное – скажем область памяти и поток для записи лога и т.п. В следующем листинге показано использование разных форм `new`.

```
1 class Widget {};
2 ...
3 Widget *w = new Widget;
4 Widget *w = new (std::nothrow) Widget;
5 Widget *w = new (WidgetPool) Widget;
6 Widget *w = new (WidgetPool, WidgetLogStream) Widget;
```

Здесь первая строчка это выделение памяти с возбуждением исключения при ошибке выделения и вызов конструктора. Вторая строчка это выделение памяти без возбуждения исключения при ошибки выделения (тогда в `w` просто запишется нулевой указатель).

Третья строчка это просто вызов конструктора на уже выделенную память (при этом неудачи выделения быть, разумеется, уже не может). И четвёртая строчка иллюстрирует специальную пользовательскую форму оператора `new` и не будет скомпилирована, без специальных объявлений внутри класса `Widget`.

2. Далее компилятор выделяет память в куче (строго при необходимости, размещающие и специальные формы пропускают этот пункт) и в случае нехватки памяти вызывает пользовательский обработчик, который можно поставить и на стандартное выделение с помощью `set_new_handler`.
3. Далее в случае если вызвана нормальная форма неразмещающего `new`, компилятор вызывает конструктор размещённого в куче объекта или все конструкторы, если был размещён массив.
4. Разрушение происходит в обратном порядке.

Обычный сценарий работы `malloc()` для выделения буфера, вместе с размещающим `new` для явного вызова конструктора приведён ниже:

```

1 class Widget {
2     int m_i;
3 public:
4     Widget(int a_i): m_i(a_i * a_i) {}
5     int get_state() const { return m_i; }
6     ~Widget() {}
7 };

1 Widget *test_malloced(int initializer) {
2     Widget *wbuf = (Widget *) malloc(sizeof(Widget));
3     Widget *w = new (wbuf) Widget(initializer);
4     return w;
5 }

1 int main() {
2     Widget *w = test_malloced(5);
3     assert(w->get_state() == 25);
4     w->~Widget();
5     free(w);
6     return 0;
7 }
```

Обратите внимание – деструктор может быть вызван явно.

### 3.4.3 Список нормальных форм

Ниже приведен класс для которого объявлены переопределения всех нормальных форм `new/delete` как это сделано в [5].

```
1 struct StandardNewDeleteForms {
2     static void *operator new(std::size_t size);
3     static void operator delete(void * memory);
4     static void *operator new(std::size_t size, const std::
5        nothrow_t &nt);
6     static void *operator new(std::size_t size, void *ptr);
7 };
```

**Домашняя наработка:** Напишите и протестируйте код для каждого из переопределений.

## 3.5 Правые ссылки и семантика перемещения

*If you take a reference to a reference to a type,  
you get a reference to that type,  
rather than some kind of special reference to reference type.*

– Bjarne Stroustrup

Ссылки (references) это то, что наиболее сложно воспринимается при переходе с C на C++. Человеку с опытом программирования на C, может быть непонятно, зачем они нужны, когда уже есть такие удобные и привычные указатели, которые позволяют делать всё то же самое только лучше... Подлинное осознание того, каким именно инструментом являются ссылки и как ими оперируют, приходит с опытом C++. При переходе на новый стандарт C++11, идея о том, что теперь есть ещё один вид ссылок – ссылки на rvalue (rvalue references) кажется новаторской и неочевидной (особенно неочевидна её полезность). Скорее всего, многим придётся преодолеть внутреннее сопротивление, прежде чем rvalue references станут удобным и привычным инструментом.

### 3.5.1 Правые и левые значения

Прежде, чем переходить к описанию rvalue references, есть смысл немного вспомнить о том, что такое rvalue и lvalue.

Термин lvalue был, прежде чем перейти в C++, документально зафиксирован в стандарте языка C, поэтому логично начать изложение с него. Для языка C, lvalue (от left hand side value) – то, что может появиться слева в выражении присваивания. Формально (6.3.2.1 стандарта C99), “An lvalue is an expression with an object type or an incomplete type other than void” (под object type в стандарте понимаются все типы, не являющиеся function type или incomplete type).

```

1 int c = a * b; // ok, c is lvalue
2 a * b = 42;   // fail, a*b is rvalue
3 foo() = 42;  // fail in C, foo() is rvalue

```

Собственно, в стандарте языка C90 и последовавшем за ним C99 нет термина rvalue. В C99 есть оговорка “What is sometimes called “rvalue” is

in this International Standard described as the “value of an expression”, позволяющая предположить, что к этому времени этот термин существовал и использовался, но фиксировать его строго в случае языка С было не нужно.

Всё изменилось с приходом языка C++, который принёс с собой ссылки (lvalue references). Выражение из прошлого примера:

```
1 foo() = 42; // in C++ this might be ok
```

может быть ошибкой если foo возвращает значение, но совершенно корректно, если foo возвращает ссылку. Чтобы разрешить эту неоднозначность, стандарт C++98 объявляет (3.10.1) два термина – lvalue и rvalue, причём каждое выражение языка C++98 является либо тем, либо другим. При этом определение в (3.10.2) гласило “An lvalue refers to an object or function”.

То есть lvalue это не первоклассный объект как в С, а просто некое выражение ссылающееся на область памяти. Таким образом в качестве главного различия выступает возможность взять адрес, а не отношение к операции равенства.

```
1 int& foo();
2 foo() = 42;      // ok
3 int* p1 = &foo(); // ok
4 int foobar();
5 int* p2 = &foobar(); // fail
```

Таким образом, lvalue в C++98 приобретает смысл locator value (а не left hand side value). Так, например если есть `int *val`, то выражение `val+1` можно разыменовать, но нельзя взять его адрес.

```
1 int *val;
2 int correct = *(val + 1); // ok
3 int **wrong = &(val + 1); // fail
```

При этом в качестве двух облачков на горизонте языка маячили проблемы семантики перемещения и проброса аргументов. Пока что эти слова достаточно упомянуть, смысл они приобретут чуть ниже по тексту. Оказалось, что обе проблемы имеют красивое решение, если дополнить язык ссылками на rvalues, также называемыми rvalue references или правыми ссылками.

### 3.5.2 Сылочное связывание

Связыванием (binding) называется сопоставление ссылки как альтернативного имени реальному содержимому памяти. Даже очень простые случаи связывания могут вызывать некоторое удивление.

```

1 int x = 1;
2 int &a = x;
3 int const &b = x;
4 int const &c = x + 1;
5 x = a + b + c;
6 cout << a << b << c; // 4 4 2

```

Многих удивляет, что здесь на экране 4 4 2, а не 4 4 5. Вроде бы ссылка `c` связана со значением `x+1`. Но что произошло на самом деле – ссылка `c` была связана с временным объектом и автоматически продлила время его жизни. Важно, что для левых ссылок так работают только константные левые ссылки и получившийся временный объект является неизменяемым без специальных хаков.

Правую (неконстантную) ссылку в её базовом виде можно считать способом связаться с изменяемым временным объектом.

```

1 int x = 1;
2 int &a = x;
3 int const &c = x + 1;
4 int &&d = x + 1;
5 c += 1; // fail
6 d += 1; // ok

```

Таким образом, в этом фрагменте кода `d` работает как ссылка на rvalue – неименованное изменяемое значение.

Несколько простых правил относительно взаимодействия и кросс-связывания ссылок изложено ниже.

1. Правая ссылка не может быть связана с lvalue

```

1 int x = 1;
2 int &&y = x * 3; // ok
3 int &&b = x; // fail, not rvalue

```

2. Неконстантная левая ссылка не может быть связана с rvalue

```

1 int &c = x * 3; // fail, not lvalue
2 const int &d = x * 3; // ok, extends lifetime

```

3. Но при этом правая ссылка сама по себе задаёт имя и адрес и является lvalue

```

1 int &&e = y; // fail, not rvalue
2 int &f = e; // ok, rvref is lvalue

```

Функция может быть перегружена по типу ссылки

```

1 int foo (int &p); // 1
2 int foo (int &&p); // 2
3 int x = 1;
4 foo (x); // 1
5 foo (x + 0); // 2
6 foo (foo (x)); // 1, then 2

```

Методы у класса могут зависеть от того является ли объект класса левым или правым значением (так же, как они различаются по сквалификаторам).

```

1 struct S {
2     int foo () &; // 1
3     int foo () &&; // 2
4 };
5 extern S bar ();
6 S x {};
7 x.foo(); // 1
8 bar().foo(); // 2

```

**Вопрос студентам:** продление времени жизни левыми константными ссылками вызывает интересные проблемы.

```

1 struct Base {};
2 struct Derived : public Base {};
3
4 Derived __attribute__((noinline))
5 derivedret ()
6 {
7     return Derived ();
8 }
9

```

```

10 void
11 tricky_1 ()
12 {
13     const Base &corr = derivedret ();
14     /* use corr */
15 } /* which destructor? */

```

**Вопрос к студентам:** Какой деструктор будет вызван в этом случае?

Теперь пришло время более интересного трюка: что если хочется связать правую ссылку со значением существующего объекта? Для этого нужно анонимизировать этот объект, превратить его в безлиное значение. Традиционно для этого применяется стандартная функция `std::move`

```

1 int x = 42;
2 int &&rvx = std::move(x);
3 rvx += 2;
4 assert(x == 44);

```

Понимание того как живут rvalue references и временные объекты, позволяет использовать класс, сочетающий левые и правые ссылки:

```

1 struct RefBind
2 {
3     int& ref;
4
5     RefBind(int&& r) : ref(r) {}
6
7     RefBind const& plus(int x) const { ref += x; return *this; }
8     RefBind const& mult(int x) const { ref *= x; return *this; }
9
10    int get() const { return ref; }
11 };

```

Три сценария использования:

```

1 int x = 1;
2
3 std::cout << RefBind(1).plus(2).mult(3).get() << std::endl;
4 std::cout << RefBind(std::move(x)).plus(2).mult(3).get() <<
    std::endl;

```

```
5 std::cout << RefBind(int(x)).plus(2).mult(3).get() << std::endl;
```

Первая строчка работает с чисто временным объектом, привязанным к lvalue ссылке и поэтому модифицируемым. Вторая попутно изменяет значение переменной `x`, третья делает из переменной временный объект и работает с ним.

Можно сказать, что `move` приводит объект к правой ссылке.

Можно ли привести объект к иной ссылке? Да, но с проблемами. Для этого в языке вроде как есть похожие функции `std::ref` и `std:: cref`. Но именно, что вроде как есть, потому что на самом деле они работают совершенно иначе.

```
1 int x = 1;
2 int &&a = move(x);
3 int &b = ref(x);
4 const int &c = cref(x);
```

С виду всё то же самое, но это симметрия обманчива, так как `std::ref` коварен:

```
1 auto f = move(x); // -> int&&
2 auto e = ref(x); // -> std::reference_wrapper<int>
```

Сылочное связывание, разумеется, задаёт уровень косвенности и эта связь может стать некорректной.

### 3.5.3 Провисание ссылок

Проблема висячих ссылок (dangling references) известна давно и каждому программисту. Для левых ссылок она была рассмотрена ранее в (2.5.2)

```
1 void simple_dangle ()
2 {
3     int *myArray;
4     myArray = new int[2]{ 100, 200 };
5     int& ref = myArray[0];
6     delete[] myArray;
7     cout << "100 = " << ref << endl; // dangling reference
8 }
```

Ссылка на что угодно, время жизни чего истекло это висячая ссылка. Но такие вещи программисты обычно отлавливают (сами или инструментально) и контролируют. Гораздо хуже, если ссылка провисает в результате ссылочного связывания при возвращении из функции.

Ниже приведено несколько примеров.

```
1● const int&
2 clref (int p) { return p + 0; } // p+0 is dead
```

Константная левая ссылка связана со времененным значением

```
1● const int&
2 clref (int p) { return p; } // p rests in peace
```

Не смотря на внешние различия, также константная левая ссылка связана со времененным значением

```
1● int&
2 lref (int p) { return p; } // p is p no more
```

Левая ссылка опять (сюрприз) связана с истекшим времененным значением

```
1● int&&
2 rref (int p) { return p + 0; } // p+0 expired and gone
```

Правая ссылка связана с истекшим времененным значением. Получается странная вещь: висячая правая ссылка.

```
1● int&&
2 rvref (int &&p) { return p + 0; } // p+0 is an ex-p+0
```

Несмотря ни на что, снова правая ссылка связана со времененным значением

На этом фоне всеобщего уныния, есть несколько на удивление работающих конструкций.

```
1● int&
2 lvref (int &x) { return x; } // ok
```

Константная левая ссылка связана со времененным значением

```

1● const int&
2 clvref (const int &x) { return x; } // also ok

```

Их общим признаком является тот факт, что идентичность объекта под ссылкой сохраняется в вызывающем фрейме.

**Вопрос к студентам:** что будет если попытаться использовать значение объекта типа RefBind (код для этого типа приведён в 3.5.2) после конца полного выражения?

```

1 RefBind x(1);
2 cout << x << endl;

```

### 3.5.4 Семантика перемещения

Первая серьёзная проблема, решение которой было обещано в начале раздела, это проблема семантического выражения перемещения. Она может быть проиллюстрирована проблемой COAP (см. 5.3.2) – действительно контейнеры из `auto_ptr` вели себя странно, потому что перемещение и копирование никак не были разделены: в языке была возможность написать `a = b` но не `move(a,b)` и заложенная в переопределенный оператор присваивания семантика перемещения оказалась миной: рано или поздно так или иначе кто-то все равно использует присваивание как присваивание.

С помощью правых ссылок, список специальных методов класса (уже включающий конструкторы, копирующие конструкторы, операторы присваивания и т.д.) был дополнен ещё двумя – перемещающими конструкторами и перемещающим присваиванием.

Перемещающий конструктор выглядит очень похоже на копирующий, но берёт правую ссылку.

```

1 class Buffer {
2     size_t sz; int *buf;
3 public:
4     Buffer (Buffer&& rhs) : sz(rhs.sz), buf(rhs.buf) {
5         rhs.sz = 0; rhs.buf = nullptr;
6     }
7     Buffer (const Buffer& rhs) : sz(rhs.sz), buf(new int[sz]) {
8         copy (rhs.buf, rhs.buf + sz, buf);
9     }

```

```
10 };
```

Если предположить, что упомянутый в листинге класс буфера содержит большое количество тяжёлых данных, видно, что перемещение выглядит куда дешевле, чем копирование.

В свете перемещающих конструкторов особенно важным становится правило, что правая ссылка это lvalue. Легко представить ситуацию, в которой в наследуемом от `Buffer` классе `DerivedBuf`, есть необходимость эти конструкторы переопределить. С первого взгляда можно сделать это следующим образом:

```
1 DerivedBuf(DerivedBuf const & rhs) : Buffer(rhs) {};
2 DerivedBuf(DerivedBuf&& rhs) : Buffer(rhs) {}; // FAIL
```

Перемещающий конструктор переопределён неверно – поскольку `rhs` имеет имя, оно является lvalue, а значит `DerivedBuf(rhs)` в данном случае вызовет снова `Buffer(Base const & rhs)`, что, вероятно, не желательно. Вместо этого следует писать

```
1 DerivedBuf(DerivedBuf&& rhs) : Buffer(std::move(rhs)) {};
```

Говорят, что `std::move` скрывает имя `rhs`, делая его таким же безымянным rvalue выражением, как `a*b`. Это общее правило “оно не должно иметь имени” очень полезно для выработки интуиции.

Аналогично перемещающему конструктору работает перемещающее присваивание

```
1 Buffer& operator= (Buffer&& rhs) {
2     if (this == &rhs)
3         return *this;
4     delete [] buf;
5     sz = rhs.sz;
6     buf = rhs.buf;
7     rhs.sz = 0;
8     rhs.buf = nullptr;
9     return *this;
10 }
```

Хороший (но не самый простой) способ показать где это нужно это рассмотреть возврат из функции по значению

```
1 Buffer foo () {
2     Buffer retval;
```

```

3 // filling retval
4 return retval;
5 }
6
7 Buffer t = foo(); // t move-constructed, not copied

```

Серьёзная ошибка это писать явный `move` внутри `return`. Это может привести к подвисанию ссылки.

Дополнительная красота этого примера в том, что не написав там ни одной правой ссылки, программист, тем не менее, активно пользуется правыми ссылками в неявном виде.

Ещё один (и пожалуй простейший) способ показать где годятся новые специальные методы это рассмотреть функцию обмена значениями (так поступают почти все известные автору источники, потому что пример уж больно нагляден).

Старый способ обмена значениями (до 2011 года)

```

1 template <typename T> void swap (T& x, T& y) {
2     T tmp = x; // copy ctor
3     x = y; // assign
4     y = tmp; // assign
5 }

```

Новый способ очевидно лучше (представьте вместо `T` класс `Buffer`).

```

1 template <typename T> void swap (T& x, T& y) {
2     T tmp = move(x); // move ctor
3     x = move(y); // move assign
4     y = move(tmp); // move assign
5 }

```

Этот код во многом вводит новичка в искушение. При взгляде на него кажется, что использование `move` само по себе действительно что-то куда-то перемещает. Вряд ли можно придумать худшее заблуждение. Кое-что, конечно перемещается, но по другой причине. Это можно проиллюстрировать следующим кодом (обратите внимание на `asserts` ниже).

```

1 int x = 1;
2 int a = move(x);
3 assert (x == a); // always ok
4 Buffer y {10};

```

```

5 Buffer b = move(y);
6 assert (y == b); // always failed

```

Идея в следующем: просто move всего лишь анонимизирует переменную ничего с ней не делая. Будет ли состояние потеряно зависит от того есть ли у класса move-конструктор и как он реализован. У int его точно нет, на чём и построен первый ответ.

Именно с тем, что настояще перемещение делает не язык, не компилятор (они всего лишь обеспечивают move специальной семантикой) а сам программист, программист может и выстрелить себе в ногу.

```

1 class SillyBuffer {
2     int sz; int *buf;
3 public:
4     SillyBuffer (int s = 10): sz(s), buf(new int[sz]){}
5     ~SillyBuffer () { delete [] buf; }
6 };
7 void swap (SillyBuffer& lhs, SillyBuffer& rhs) {
8     SillyBuffer tmp = move (lhs);
9     lhs = move (rhs);
10    rhs = move (tmp);
11 } // UB (probably segfault)

```

Перемещение по умолчанию перемещает по умолчанию все поля класса. Перемещая по умолчанию указатель оно побитово его копирует. После этого столь глупое перемещение будет вести себя не лучше, чем копирование по умолчанию для того же класса.

Классическая идиома проектирования rule of five утверждает, что если ваш класс требует нетривиального определения хотя бы одного из пяти методов:

- копирующего конструктора
- копирующего присваивания,
- перемещающего конструктора
- перемещающего присваивания
- деструктора

то вам лучше бы нетривиально определить все пять

Очевидно `SillyBuffer` его нарушает: он определяет нетривиальный деструктор и только его.

Итак, хорошо организованный move ctor изменяет rhs. Но что если rhs нельзя изменить?

```
1 const Buffer y {10};
2 Buffer b = move(y);
3 assert (y == b);
```

В этом случае move ctor просто не будет вызван, так как его сигнатура предполагает `Buffer &&` а не `Buffer const &&`.

Вместо этого, `Buffer const &&` будет приведён к `Buffer const &` и вызовется копирующий конструктор, несмотря на явное указание `move`.

Таблица специальных методов подытоживает этот раздел (теперь вы знаете достаточно, чтобы понимать здесь каждую строчку). Любой специальный метод, не объявленный пользователем, может быть объявлен по умолчанию (def), отсутствовать (none) или стёрт (del). Крестики стоят там, где строка, соответствующая методу объявлению пользователем, совпадает со столбцом, где перечислены неявные методы.

|              | default ctor | dtor | copy ctor | copy assign | move ctor | move assign |
|--------------|--------------|------|-----------|-------------|-----------|-------------|
| Nothing      | def          | def  | def       | def         | def       | def         |
| Any ctor     | none         | def  | def       | def         | def       | def         |
| Default ctor | x            | def  | def       | def         | def       | def         |
| Dtor         | def          | x    | def       | def         | none      | none        |
| Copy ctor    | none         | def  | x         | def         | none      | none        |
| Copy assign  | def          | def  | def       | x           | none      | none        |
| Move ctor    | none         | def  | del       | del         | x         | none        |
| Move assign  | def          | def  | del       | del         | none      | x           |

Конечно, главная проблема здесь в обратной совместимости с C++98 в том, что касается копирования и присваивания по умолчанию даже если пользователь определил только один из этих двух методов. Это был суровый просчёт в проектировании и сколько лет ещё с ним жить – не очень понятно. Благо к этим конкретным граблям все давно привыкли.

Здесь заканчивается тонкая настройка “волшебных очков” выданных ранее в (3.2.2). Теперь через них видно всё.

**Вопрос студентам:** какую из двух предложенных сигнатур вы выберете и почему?

```

1 vector<string> create_vector_of_strings(); // 1
2 create_vector_of_strings(vector<string>&); // 2

```

Итак, rvalue references позволяют (совместно с `std::move`) по новому взглянуть на ссылки и открывают простор к более тонкому различию таких терминов естественного языка как “копирование”, “присваивание значения”, “присваивание результата” на языке C++, что открывает как простор к оптимизациям, так и возможности более аккуратного выражения старых идиом (хороший пример блестящего использования правых ссылок это `std::unique_ptr` который в новом стандарте пришёл на смену печально известному `auto_ptr`).

```

1 auto_ptr <int> told (new int(2));
2 auto_ptr <int> sold (new int(3));
3
4 told = sold;
5
6 unique_ptr <int> t (new int(2));
7 unique_ptr <int> s (new int(3));
8
9 // t = s; -- compilation error, no copy ctor
10 t = std::move (s); // -- ok, move ctor

```

Здесь правило большой пятерки вовсе не нарушено – копирующее присваивание в `unique_ptr` сознательно запрещено через `=delete`, а это вполне нетривиальное определение. Больше про умные указатели нового стандарта см. (5.3).

Это закрывает первый вопрос и теперь настало время поговорить о втором, но сначала необходимо узнать больше про правила работы с правыми и левыми ссылками.

### 3.5.5 Свёртка ссылок

Два вида ссылок дают возможность развернуться с записью ссылок на ссылки на ссылки. Простейший случай вывода типов уже ставит соответствующие проблемы.

```

1 int x;
2 int &y = x;
3 auto &&c = y; // -> yields what?

```

Уточнённое с помощью rvalue reference, auto не может игнорировать ссылку. Формально вывод выглядит так:

```
1 auto &&c = y; // -> int & && c = y;
2 auto &&d = move(y); // -> int && && c = y;
```

Чтобы получился корректный тип, ссылки должны быть свёрнуты (collapsed).

Стандарт C++11 (8.3.2.6) определяет следующие правила свёртки ссылок, применимые для определений `auto` и `decltype`, `typedef`, а также параметров шаблонов:

`A& &` становится `A& A& &&` становится `A& A&& &` становится `A& A&& &&` становится `A&&`

Говоря коротко, левая ссылка выигрывает там, где она есть. Для рассмотренного выше примера это даёт:

```
1 auto &&c = y; // -> int & && c -> int &c = y;
2 auto &&d = move(y); // -> int && && d
3 // -> int &&d = move(y);
```

Правила вывода дают интересную картину: `auto&` это всегда lvalue ref, но `auto&&` это либо lvalue ref, либо rvalue ref (зависит от контекста)

```
1 auto &&y = x; // x это some& -> y это some&
```

Это в целом работает и для `decltype`

```
1 decltype(x) &&z = x; // the same
```

И для шаблонов

```
1 template <typename T> void foo(T&& t);
2 foo(x); // the same
```

Такие ссылки называют forwarding references. Майерс [7] предложил называть их универсальными ссылками. По русски “универсальные” звучит лучше, чем “пробросовые” поэтому неформальный термин Майерса я буду употреблять чаще.

Важно отличать настоящие универсальные ссылки от фальшивых (от последних никакой радости).

Контекст сворачивания требует вывода типов, а не их подстановки:

```
1 template<typename T> struct Buffer {
```

```

2   void emplace (T&& param); // T is substituted
3   ...
4 };

```

В примере выше параметр известен в момент инстанцирования класса и просто подставляется в метод. Это правая ссылка, никакого сворачивания тут нет. А вот в примере ниже оно появляется.

```

1 template<typename T> struct Buffer {
2   template<typename U>
3     void emplace (U&& param); // U is inferred and collapsed
4   ...
5 };

```

Контекст для сворачивания не будет создан, если тип уточнён более, чем `&&`

```
1 template<typename T> void buz (const T&& param);
```

Это так же верно для `auto`

```
1 const auto &&x = y; // rvalue ref
```

### 3.5.6 Пробрасывание, близкое к совершенству

Название perfect forwarding несколько излишне амбициозно. Я предпочитаю переводить его на русский язык, как “проброс, близкий к совершенству”. Постановка проблемы следующая: некая функция принимает аргумент и должна, ничего с ним не сделав, передать его как аргумент другой функции. При этом если аргумент изначально был `rvalue`, он должен прийти как `rvalue`, а если он был `lvalue`, то как `lvalue`. Это и есть вторая серьёзная проблема из заголовка. Полностью совершенным этот проброс назвать нельзя, так как категорий значений на самом деле не две, а пять (см. дальше) но по условию задачи должны сохраняться только две (реально сохраняются четыре).

Чтобы проиллюстрировать актуальность, можно рассмотреть прозрачную оболочку (первое приближение см. `transparent` в разделе 2.11.4).

```

1 template <typename Fun, typename Arg> decltype(auto)
2 transparent (Fun fun, Arg arg) {
3   return fun(arg);
4 }

```

Увы, её недостаток в том, что она не слишком прозрачна. Например следующий код очевидно порождает лишнее копирование при создании временной переменной для формального параметра.

```
1 extern Buffer foo (Buffer x);
2 Buffer b;
3 Buffer t = transparent (&foo, b); // copy b
```

Возможный выход: уточнить аргумент ссылкой. Пока непонятно какая подойдёт лучше, но с примером выше будет работать левая, так что можно попробовать левой.

```
1 template <typename Fun, typename Arg> decltype(auto)
2 transparent (Fun fun, Arg& arg) {
3     return fun(arg);
4 }
```

Теперь лишнего копирования для rvalues вроде бы нет. Но появляется новая беда: теперь rvalues не проходят в функцию.

```
1 extern Buffer foo (Buffer x);
2 Buffer b;
3 Buffer t = transparent (&foo, b); // ok
4 Buffer u = transparent (&foo, foo(b)); // fail, CE
```

И действительно, левая ссылка не может быть связана с rvalue. Возможный выход: перегрузить по константной ссылке.

```
1 template <typename Fun, typename Arg> decltype(auto)
2 transparent (Fun fun, Arg& arg) { return fun(arg); }
3
4 template <typename Fun, typename Arg> decltype(auto)
5 transparent (Fun fun, const Arg& arg) { return fun(arg); }
```

Теперь вроде всё хорошо. Идея неплоха и в общем в C++98 все так и делали.

```
1 Buffer t = transparent (&foo, b); // ok
2 Buffer u = transparent (&foo, foo(b)); // ok, but copy
```

Но есть проблемы:

1. Всего 10 аргументов потребуют 1024 перегрузки
2. Вызов для rvalue всё ещё требует копирования

Решение для первой проблемы: универсализовать ссылку

```

1 template <typename Fun, typename Arg> decltype(auto)
2 transparent (Fun fun, Arg&& arg) {
3     return fun(arg);
4 }
```

Увы, даже с универсализованной ссылкой, вызов для `rvalue` всё ещё требует копирования. Если подумать, то не хватает единственного нюанса: возможности внутри функции определять категорию значения аргумента. Следующий листинг иллюстрирует эту идею в качестве *wishful thinking*. Это не корректный C++11 но я не могу поручиться, что этот код всегда останется некорректным.

```

1 // This is NOT C++11
2 template <typename Fun, typename Arg> decltype(auto)
3 transparent (Fun fun, Arg&& arg) {
4     if (arg is rvalue)
5         return fun(move(arg));
6     else
7         return fun(arg);
8 }
```

Это решило бы все проблемы. Подобная реализация возможна с помощью `std::forward`, который в общем делает именно это.

```

1 template <typename Fun, typename Arg> decltype(auto)
2 transparent (Fun fun, Arg&& arg) {
3     return foo(forward<Arg>(arg));
4 }
```

Теперь и левые и правые значения прозрачно проходят (пробрасываются) через оболочку к месту использования. Решение проблемы достигается взаимодействием трёх синтаксических конструкций, которые я называю “шестерёнками perfect forwarding”.

1. Проброс возможен только в **контексте вывода типа**. Здесь эту роль играет `typename Arg` в шаблоне функции.
2. Проброс возможен только при наличии универсальной ссылки, в данном случае это `Arg&&` в сигнатуре
3. Для проброса нужна конструкция `forward<Arg>` а параметр должен быть тот самый который пришёл под универсальной ссылкой на аргумент.

Простые случаи проброса очень просто описать и понять.

```

1 void g(int &&t) { cout << "r"; }
2 void g(int &t) { cout << "l"; }
3
4 template <typename T> void
5 h(T &&t) { g(forward<T>(t)); }
```

Здесь очевидно, что функция `h` вызывает правильную функцию `g`.

```

1 h(0); // prints "r"
2 int x = 0;
3 h(x); // prints "l"
4 h(x + 0); // prints "r"
```

Но сложные случаи иногда оказываются хм... сложны.

```

1 const int x = 1;
2 h(x); // ???
```

Чтобы отвечать на такие вопросы, нужно понимать как работают `move` и `forward` на чуть более глубоком уровне. В свою очередь чтобы достичь этого понимания, необходимо научиться базовой работе с типами. Эта работа будет рассмотрена подробно в (4.8.3), а сейчас будет только небольшой тизер.

Объекты, которые ведут себя как функции, называются функторами. Функторы бывают над данными или над типами. Полезный функтор над типом `T` который записывается как `remove_reference_t<T>` очищает этот тип от ссылок если они там были (или ничего не делает, если их там не было).

```

1 int x = 42;
2 int &a = x;
3 int &&b = move(x);
4 remove_reference_t<decltype(a)> s; // -> int s;
5 remove_reference_t<decltype(b)> u; // -> int u;
6 remove_reference_t<decltype(x)> v; // -> int v;
```

Теперь можно привести код, который соответствует возможной реализации `std::move`.

```

1 template <typename T> decltype (auto)
2 almost_move(T&& a) {
3     using rv_ref_t = remove_reference_t<T>&&;
```

```

4     return static_cast<rv_ref_t>(a);
5 }
```

Здесь `using` это вариант `typedef` для работы в шаблонных контекстах, см. (4.2.9).

Первое, что тут бросается в глаза – `move` это просто `static_cast` ничего больше. Кроме того здесь есть контекст вывода типа и именно поэтому `move` почти всегда употребляется без уточнения типа.

Очень похоже устроен `forward`

```

1 template<typename S> S&&
2 almost_forward(remove_reference_t<S>& a) {
3     return static_cast<S&&>(a);
4 }
```

Но здесь вывод типа убит формой аргумента, поэтому для `forward` всегда указывают управляющий параметр.

Теперь ответ на поставленный выше вопрос можно вычислить вручную размотав работу проброса.

```

1 const int x = 1;
2 h(x); // ???
```

Экземпляр `forward` с подставленным типом будет выглядеть так:

```

1 const int&
2 almost_forward<const int&> (const int& a) {
3     return static_cast<const int&>(a);
4 }
```

Таким образом функция `g` будет вызвана с параметром типа `const int`, ну а поскольку такой перегрузки не было определено, будет ошибка компиляции.

**Вопрос к студентам:** что будет если нарушить форвардинг и использовать `move` вместо `forward`?

```

1 void g(int &&t) { cout << "r"; }
2 void g(int &t) { cout << "l"; }
3 template <typename T> void h(T &&t) { g (move(t)); }
4 int x = 42;
5 h(x); // ???
```

Например что произойдёт в этом участке кода?

Это подытоживает вторую серьёзную проблему. Теперь до конца раздела осталось одно небольшое философское обсуждение, которое можно пропустить и перейти к наследованию.

### 3.5.7 Категории значений

По сути весь этот раздел речь шла о категориях значений, таких как rvalue и lvalue. Но откуда берутся эти категории, что они на самом деле означают и сколько их на самом деле?

Здесь будут обсуждаться значения как таковые.

В некоторых случаях (например для использования внутри `decltype`) хочется получить значение некоего типа. Часто для этого используется конструктор по умолчанию. Но что делать, если его нет? Что такое “значение вообще” для такого типа?

```

1 template <typename T> struct Tricky {
2     Tricky() = delete;
3     const volatile T foo ();
4 };
5
6 decltype(Tricky().foo()) t; // fail

```

Человек, искушённый в С мог бы сказать, что значение как таковое это результат приведения сырой памяти

```

1 template <typename T> struct Tricky {
2     Tricky() = delete;
3     const volatile T foo ();
4 };
5
6 decltype(((*Tricky)0)->foo()) t; // works, but...

```

К сожалению, это слишком хрупко. Если такое абстрактное значение проникнет в runtime контекст, это будет runtime ошибка.

Интересный способ решить эти проблемы это ввести шаблон функции (который выводит типы) без тела (чтобы его нельзя было по ошибке вызвать).

```
1 template <typename T> add_rvalue_reference_t<T> declval();
```

Здесь `add_rvalue_reference_t<T>` это ещё один полезный функтор, который, как нетрудно догадаться, добавляет правую ссылку к произвольному типу. Теперь всё просто.

```

1 template <typename T> struct Tricky {
2     Tricky() = delete;
3     const volatile T foo ();
4 };
5
6 decltype(declval<Tricky>().foo()) t; // ok

```

Но какова природа этого значения? Оно не rvalue и не lvalue.

Для того, чтобы осознать это, необходимо понимать фундаментальное различие между категориями значений: различие по идентичности и по перемещаемости.

- Объект обладает идентичностью если у него есть имя и адрес в памяти
- Объект обладает перемещаемостью, если его состояние по использованию не важно.

Например

```

1 extern int foo ();
2 int x = foo ();

```

Здесь `x` обладает идентичностью, временный результат `foo` перемещаем. Начиная с 2011 года объект может обладать и тем и другим, за счёт появления правых ссылок.

```

1 Buffer d;
2 Buffer b = move (d);

```

Итоговая диаграмма приведена на (рис. 3.2)

Здесь объекты, обладающие идентичностью называются glvalues (или generalized lvalues), а обладающие перемещаемостью это rvalues. Перемещаемость без идентичности это prvalues, а идентичность без перемещаемости это lvalues. И, наконец, самые странные штуки в середине, обладающие и тем и другим это xvalues.

`xvalue` (от expiring value) это значение, возвращённое из функции по правой ссылке, например `move(x)` это `xvalue`.

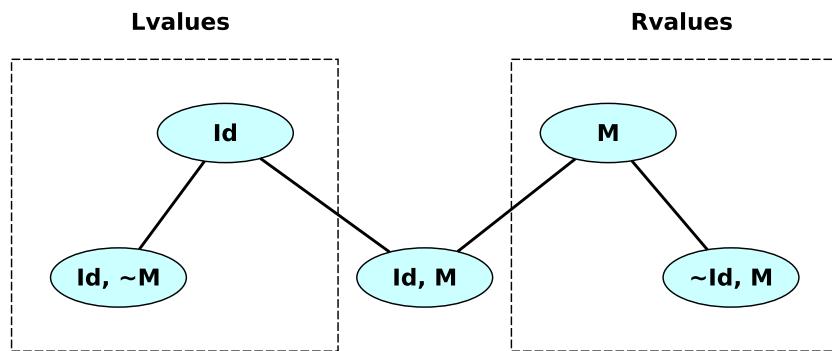


Рис. 3.2: Идентичность и перемещаемость

Стандарт предлагает функторы-определители.

- `is_rvalue_reference<T>::value` определяет является ли тип `xvalue`
- `is_lvalue_reference<T>::value` определяет является ли тип `lvalue`

Например

```

1 int x = 42;
2 assert(is_lvalue_reference<decltype(x)>::value);
3 assert(is_rvalue_reference<decltype(move(x))>::value);

```

Имея только два эти функтора, остальные категории значений можно легко определить используя их комбинации.

```

prvalue = ~xvalue & ~lvalue
glvalue = xvalue & lvalue
rvalue = xvalue & ~lvalue

```

Пожалуй есть всего три функции, для которых имеет смысл возвращать правую ссылку (то есть производить `xvalue result`)

- `std::move`
- `std::forward`
- `std::declval`

Если вы хотите написать свою функцию, которая будет возвращать `&&`, это значит, что вы что-то делаете не так, или вы хотите ещё раз написать одну из упомянутых выше функций.

## 3.6 Наследование интерфейса

*If you think C++ is not overly complicated, just what is  
a protected abstract virtual base pure virtual private destructor?  
And when was the last time you needed one?*

– Tom Cargill

Программа, в которой важной абстракцией является физический объект, иногда подразумевает сложные архитектурные решения, относительно представления этого объекта. Координаты нужны (почти) всем физическим объектам, но как быть с массой, с радиусом, с электрическим зарядом, с любыми дополнительными параметрами? Конечно можно написать один тип со всем богатством возможностей внутри него

```

1 class AnyPhysicalBody
2 {
3     private:
4         double x, y, z;
5         double vx, vy, vz;
6         double m;
7         double r;
8     /* ... */
9     public:
10     int move (double dt);
11 };

```

Но создавать объект такого типа, когда нужна всего лишь материальная точка с массой – означает тащить за собой массу ненужных полей с произвольными и не нужными значениями. Это называется overhead – пустая трата памяти и машинного времени. Вторая возможность – написать небольшие конкретные типы на каждый случай:

```

1 class CelestialBody
2 {
3     double x, y;
4     double vx, vy
5     double m;
6 public:
7 /* ... */
8     int move (double dt);
9     int apply_force (double f, double dt);

```

```

10  };
11
12 class Planet
13 {
14     double x, y;
15     double vx, vy
16     double m;
17     double r;
18 public:
19 /* ... */
20     int move (double dt);
21     int apply_force (double f, double dt);
22     int detect_collision (const Planet *rhs) const;
23     int detect_collision (const CelestialBody *rhs) const;
24 };

```

Это работает, но возникают две проблемы: методы `detect_collision` внутри класса `Planet` дублируются (это не страшно, но может быть неприятно) и гораздо более печальная проблема – принципиально разные типы `Planet` и `CelestialBody` теперь не могут быть объединены, скажем в массив указателей на небесные тела. Решение этой проблемы состоит в том, чтобы указать компилятору на **общий интерфейс** или на то, что планета тоже является небесным телом (и материальной точкой).

### 3.6.1 Чисто виртуальные функции и абстрактные классы

Интерфейсным или чисто абстрактным является пользовательский тип, в котором определён хотя бы один **чисто виртуальный** метод. Ключевое слово `virtual` предусматривает довольно сложное поведение и будет подробно рассмотрено в (3.6.3). Чисто виртуальные методы легко различать по спецификации `=0`. Хорошим тоном является писать тип, полностью состоящий из таких методов.

```

1 struct ICelestialBody
2 {
3     virtual double get_x () const = 0;
4     virtual double get_y () const = 0;
5     virtual int move (double dt) = 0;
6     virtual int apply_force (double f, double dt) = 0;
7 };

```

Объекты такого типа нельзя создавать, его методы не могут иметь реализации, поэтому такой класс называется **абстрактным**. Это общий интерфейс – обещание того, что все реализующие его пользовательские типы реализуют две именно такие функции. Такое обещание пользовательский тип может дать через механизм открытого наследования интерфейса.

```
1 class CelestialBody: public ICelestialBody
2 {
3     double x, y;
4     double vx, vy;
5     double m;
6     public:
7     /* ... */
8     /* ICelestialBody implementation */
9     double get_x () const { return x; }
10    double get_y () const { return y; }
11    int move (double dt);
12    int apply_force (double f, double dt);
13    /* ... */
14};
```

Синтаксис открытого наследования очевиден из примера. Указание модификатора `public` обязательно, поскольку наследование может быть и иным см. (3.7.4). Для выражения идеи о “реализации интерфейса” подходит только открытое наследование.

```
1 class Planet: public ICelestialBody
2 {
3     double x, y;
4     double vx, vy;
5     double m;
6     double r;
7     public:
8     /* ... */
9     /* ICelestialBody implementation */
10    double get_x () const { return x; }
11    double get_y () const { return y; }
12    int move (double dt);
13    int apply_force (double f, double dt);
14    int detect_collision (const ICelestialBody &rhs) const;
15};
```

Разумеется, каждый класс может открыто наследовать от любого количества интерфейсов, но на данном этапе имеет смысл ограничиться одиночным наследованием, отложив рассмотрение обобщений до (3.8).

Визуально открытое наследование интерфейса представляет UML-диаграммой, показанной на (рис. 3.3).

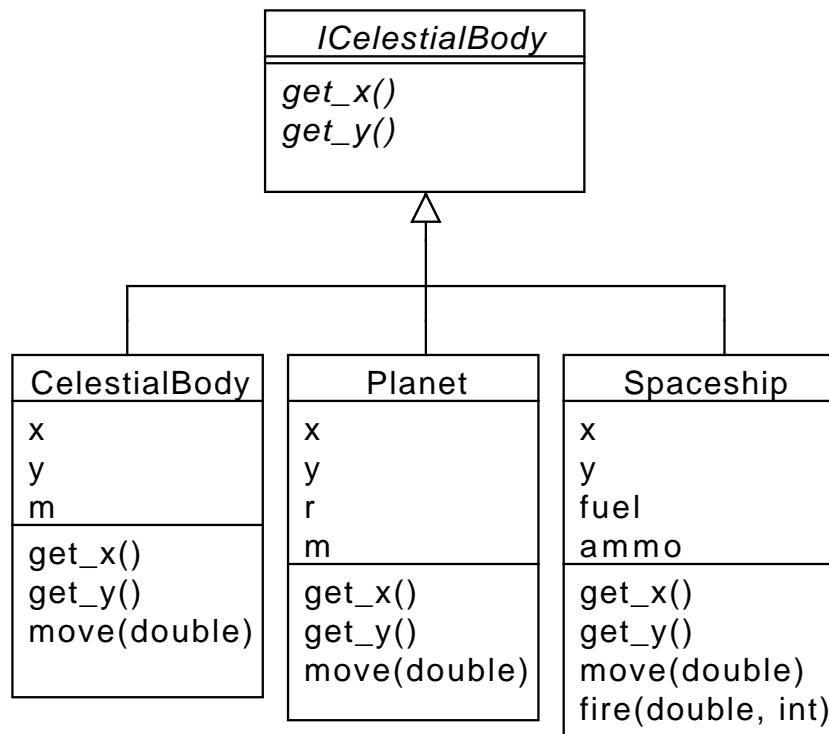


Рис. 3.3: Открытое наследование интерфейса

Теперь можно писать довольно абстрактный код, использующий приведение указателя или ссылки на объект, к указателю или ссылке на его открытый интерфейс. Как обычно сначала написать небольшую служебную функцию:

```

1 int
2 in_between (double x, double ymin, double ymax)
3 {
4     return (x <= ymax) && (x >= ymin);
5 }
  
```

И определить

```

1 int
  
```

```

2 Planet::detect_collision (const ICelestialBody &rhs) const
3 {
4     return in_between (rhs.get_x(), x - r, x + r) &&
5         in_between (rhs.get_y(), y - r, y + r);
6 }
```

Совершенно неважно с чем может произойти столкновение, если это “что-то” поддерживает все методы ICelestialBody:

```

1 Planet Jupiter;
2 Planet Earth;
3 CelestialBody Gallea;
4 /* ... */
5
6 Earth.detect_collision(Jupiter);
7 Earth.detect_collision(Gallea);
```

Это похоже на общий разъём розетки, куда в стандартный интерфейс можно подключить много разных устройств (рис. 3.4).

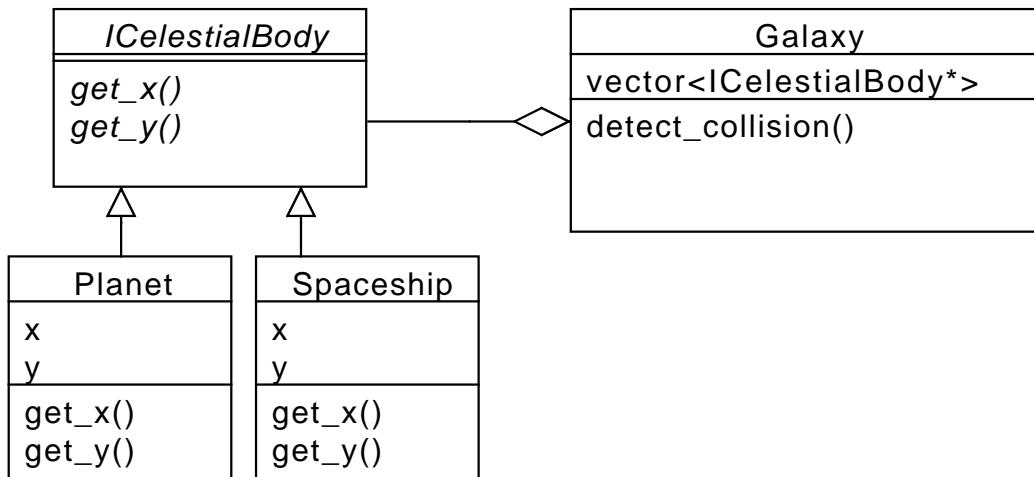


Рис. 3.4: Открытый интерфейс как общий разъём для подключения

Аргумент rhs передан по константной ссылке. В принципе, он мог быть передан и по обычной ссылке и по (константному) указателю:

```

1 int
2 Planet::detect_collision (const ICelestialBody *rhs)
3 {
4     return in_between (rhs->get_x(), x - r, x + r) &&
```

```

5      in_between (rhs->get_y(), y - r, y + r);
6  }
```

По понятным причинам в случае абстрактного базового класса, аргумент не может быть передан по значению, так как не может быть сконструирован объект такого типа.

### 3.6.2 Статический и динамический тип

Функция `Planet::detect_collision` вызывает внутри себя виртуальную функцию `ICelestialBody::get_x()` которая имеет один неявный аргумент – указатель `this` того объекта, для которого она вызвана. Этот указатель протягивается через формальный аргумент.

Что можно сказать об аргументе `rhs`? Всего лишь, что его тип, объявленный в списке аргументов (иначе говоря статический тип) это константная ссылка на `ICelestialBody`. Это всё, что компилятор может сказать об этом типе **статически**, без контекста исполнения.

Но **динамически**, при вызове из `main`, функция `detect_collision` вызывается из двух разных мест с двумя разными параметрами, один из которых имеет тип `Planet`, другой – тип `CelestialBody` и именно эти типы имеет указатель `this` для которого вызывается `get_x()` в первом и во втором случае.

При вызове функции, она должна быть верно связана с именем типа, поскольку имя типа участвует в манглированном имени функции, как это объяснялось в (2.6). При обычном вызове функции или метода, она связывается со своим типом статическим или ранним связыванием. Динамическое или позднее связывание работает при вызове виртуальной функции по ссылке или указателю на интерфейс. При этом решение о том какая функция в действительности будет вызвана, откладывается до момента вызова и принимается на основании её динамического типа. Обычно это требует дополнительного уровня косвенности для вызова через таблицу виртуальных методов, которая будет рассмотрена в (3.6.3).

Если при динамическом связывании, функция имеет аргумент, который может принимать значения разных динамических типов с которыми она вызвана, этот аргумент называется **полиморфным** аргументом для динамического полиморфизма, а сама функция – полиморфной (динамически полиморфной) функцией. В случае любой виртуальной функции единственный её полиморфный аргумент – неявный указатель на объект,

для которого она вызвана.

Вообще говоря, динамический тип может быть разрешён в любой базовый класс, в том числе и в абстрактный. При этом производится попытка прямого вызова чисто виртуальной функции. Это ошибка, но язык C++ всегда даёт программисту возможность выстрелить себе в ногу, это фирменный стиль. Поэтому важно знать тот случай, когда это наиболее часто может произойти.

### 3.6.3 Подробнее о чисто виртуальных функциях

Чтобы понять, что именно делает компилятор для того, чтобы вызов по указателю на интерфейс делегировался верному динамическому типу, полезно попробовать написать нечто подобное на С.

Сначала определяется таблица виртуальных методов

```

1 typedef double (get_t)(void *);
2
3 struct celestial_body_vmt
4 {
5     get_t *pget_x;
6     get_t *pget_y;
7     /* ... */
8 };

```

Дальше указатель на эту таблицу идёт в структуру интерфейса:

```

1 struct icelestial_body
2 {
3     struct celestial_body_vmt vtable;
4     /* ... */
5 };

```

Инициализация VMT происходит в конструкторе интерфейса:

```

1 void
2 icelestial_ctor(struct icelestial_body *ths, get_t *pget_x,
3     get_t *pget_y)
4 {
5     ths->vtable.pget_x = pget_x;
6     ths->vtable.pget_y = pget_y;
7     /* ... */
8 }

```

Каждый класс-наследник содержит в себе интерфейсную часть и собственную реализацию интерфейсных функций:

```

1 struct celestial_body
2 {
3     struct icelestial_body intf;
4     double x, y;
5     /* ... */
6 };
7
8 double celestial_get_x(void *ths) { return ((struct
9     celestial_body *)ths)->x; }
10 double celestial_get_y(void *ths) { return ((struct
11     celestial_body *)ths)->y; }
```

Конструктор класса-наследника устанавливает известные ему функции в таблицу виртуальных функций интерфейсного класса:

```

1 void
2 celestial_ctor(struct celestial_body *ths)
3 {
4     icelestial_ctor (&ths->intf, &celestial_get_x, &
5                     celestial_get_y);
6     /* ... */
}
```

Теперь всё готово для того, чтобы правильно сконструированный объект наследника мог участвовать в обобщённом коде наподобие уже рассмотренного collision detection (для простоты можно не рассматривать детали планет, имеющих некий радиус, а принять столкновение двух небесных тел как прохождение их на расстоянии epsilon):

```

1 int
2 detect_collision (struct icelestial_body *lhs, struct
3                     icelestial_body *rhs)
4 {
5     double x = lhs->vtable.pget_x(lhs);
6     double y = lhs->vtable.pget_y(lhs);
7
8     return in_between (rhs->vtable.pget_x(rhs), x - epsilon, x +
9                        epsilon) &&
10            in_between (rhs->vtable.pget_y(rhs), y - epsilon, y +
11                        epsilon);
```

```
9 }
```

Вызов этого кода требует конструирования (вручную, поскольку используется С). Крайне многословный код, но так бывает всегда в недосахаренных системах.

```
1 struct celestial_body Gallea;
2 struct celestial_body Encke;
3
4 celestial_ctor (&Gallea);
5 celestial_ctor (&Encke);
6
7 /* ... */
8
9 detect_collision (&Gallea.intf, &Encke.intf);
```

Таким образом, очевидно насколько много компилятор C++ прячет под капот. Знание механики работы виртуальных функций необязательно, но часто и очень сильно помогает. Например теперь ясно, что если бы передача параметров внутрь `detect_collision` шла по значению, то они должны были быть сконструированы внутри функции. Но поскольку у `icelestial_body` нет собственных координат и нет методов, чтобы их вернуть, то единственное, что можно передать в такой конструктор это нули:

```
1 int
2 detect_collision (struct icelestial_body lhs, struct
3     icelestial_body rhs)
4 {
5     double x, y;
6
7     icelestial_ctor (&lhs, 0, 0);
8     icelestial_ctor (&rhs, 0, 0);
9
10    x = lhs->vtable->pget_x();
11    y = lhs->vtable->pget_y();
12
13    return in_between (rhs->vtable->pget_x(), x - epsilon, x +
14                      epsilon) &&
15                      in_between (rhs->vtable->pget_y(), y - epsilon, y +
16                      epsilon);
17 }
```

И тогда вызов чисто виртуальной функции превращается в аналог вызова функции по нулевому указателю. При передаче же по указателю происходит всего лишь его приведение к указателю другого типа, без потери данных. Подобным образом работает и передача по ссылке в C++. Есть ещё несколько поучительных уроков: наличие дополнительного расхода памяти в каждом объекте на таблицу виртуальных функций, наличие дополнительной косвенности по вызову, ухудшение возможностей компилятора для онлайн-подстановки довольно простых методов и так далее – всё это очевидно при понимании того, как вещи функционируют на самом деле.

Теперь самое время разобрать несколько хитрых вопросов, относящихся к наследованию интерфейса:

### 3.6.4 Параметры по умолчанию

Тонкий вопрос, который многие программисты на C++ часто упускают из виду это то, как ведут себя виртуальные функции с параметрами по умолчанию. Если параметр по умолчанию в интерфейсе изменён в одном из его наследников:

```
1 struct ICelestialBody
2 {
3     /* ... */
4     virtual int apply_force (double f, double dt = 0.1) = 0;
5 };
6
7 void
8 forcesApply (ICelestialBody *pbody, double *pfs, int n)
9 {
10     for ( ; n > 0; --n)
11         pbody->apply_force(pfs[n - 1]);
12 }
13
14 class CelestialBody: public ICelestialBody
15 {
16     public:
17     /* ... */
18     int apply_force (double f, double dt = 0.01);
19 }
```

После чего этот обобщённый код вызван для конкретного динамического типа:

```
1 double fs[5] = {1.0, 1.5, 1.2, 1.8, 1.10};  
2 CelestialBody t;  
3 forcesApply (&t, fs, 5);
```

Казалось бы – реально будет вызвана функция `apply_force` из класса `CelestialBody` и логично ожидать значение аргумента по умолчанию таким, как он выставлен там. Но в реальности эта функция будет вызвана со значением аргумента по умолчанию, выставленным в интерфейсе.

Дело в том, что функции в C++ связываются динамически, а аргументы по умолчанию – статически. Если вернуться к сициальному аналогу, можно легко увидеть, что в таблице виртуальных функций нет места для хранения и перезаписи аргументов по умолчанию.

### 3.6.5 Виртуальные деструкторы

Для программиста на C++ важно знать один очень специальный случай полиморфизма, относящийся к семантике освобождения, а именно виртуальные деструкторы. Рассмотрим пример – пусть у нас есть фабричная функция, которая в зависимости от радиуса желаемого пользователем небесного тела возвращает `CelestialBody` если он меньше порогового значения или планету в другом случае:

```
1 ICelestialBody *get_celestial_body (double r) {  
2     if (r < epsilon)  
3         return new CelestialBody ();  
4  
5     return new Planet ();  
6 }
```

Что произойдёт если планета была получена через эту функцию, а потом удалена из памяти обычным способом?

```
1 ICelestialBody *planet = get_celestial_body (100.0);  
2 /* ... */  
3 delete planet;
```

Будет освобождена базовая часть, но не будет освобождена производная часть, что неминуемо приведёт к утечкам памяти и проблемам.

Решение этой проблемы: сделать деструктор виртуальным в интерфейсе. Тогда будет автоматически правильно вызван деструктор производного класса по указателю на базовый.

```

1 struct ICelestialBody
2 {
3     virtual double get_x () const = 0;
4     virtual double get_y () const = 0;
5     virtual int move (double dt) = 0;
6     virtual int apply_force (double f, double dt) = 0;
7     virtual ~ICelestialBody() {}
8 };

```

Наличие в классе невиртуального деструктора является в мире промышленного программирования достаточным основанием никогда ничего не наследовать от этого класса. Обратное – дурной тон.

### 3.6.6 Тела для чисто виртуальных функций

**Вопрос к студентам:** можно ли объявить деструктор чисто виртуальным следующим образом:

```

1 struct ICelestialBody
2 {
3     /* .... */
4     virtual ~ICelestialBody() = 0;
5 };

```

Тем не менее, иногда чисто виртуальный деструктор имеет смысл – например если это единственная функция в интерфейсе. Выход: написать тело для чисто виртуального деструктора

```

1 virtual ICelestialBody::~ICelestialBody()
2 {
3 }

```

В том случае, если у чисто виртуальной функции существует тело, эта функция всё ещё не может быть вызвана как метод своего класса, но она может быть вызвана из любого своего наследника.

**Вопрос к студентам:** как вы думаете зачем ещё может быть использована эта техника?

### 3.6.7 Виртуальные конструкторы

Статическая функция не может в общем случае быть ни виртуальной ни (тем более) чисто виртуальной. Поэтому конструкторы тоже не могут быть виртуальными.

**Вопрос к студентам:** а нужны ли нам вообще виртуальные конструкторы?

На этом нужно остановиться особо. Было упомянуто (см. 3.2.1), что для RAII фундаментальные переопределяемые функции это конструктор копирования и оператор присваивания. И вдруг оказывается, что второй может быть виртуальным, а первый никак:

```

1 class IVector
2 {
3 public:
4     virtual IVector& operator= (const IVector &rhs) = 0; // OK
5     virtual IVector(const IVector *rhs) = 0; // ERROR
6 };

```

Приходится выкручиваться. Обычно в рамках выкручивания люди заводят виртуальную функцию `clone` для получения копии объекта производного класса по указателю на базовый класс:

```

1 class IVector
2 {
3 public:
4     virtual IVector* clone() = 0; // virtual ctor idiom
5     virtual void operator+= (const IVector &lhs) = 0;
6 };
7
8 class Vector : public IVector
9 {
10 public:
11     Vector (const Vector &rhs); // copy ctor
12     void operator+= (const IVector &lhs); // add any vector to
13         given
14     IVector *clone() {
15         return new Vector(*this);
16     }
17 };

```

```
18 class SparceVector : public IVector
19 {
20 public:
21     SparceVector (const SparceVector &rhs); // copy ctor
22     void operator+= (const IVector &lhs); // add any vector to
23     given
24     IVector *clone() {
25         return new SparceVector(*this);
26     }
26 };
```

Теперь общий код для сложения, которое поддерживает как обычные так и разреженные вектора можно реализовать в терминах виртуальных конструкторов:

```
1 IVector *add (const IVector &rhs, const IVector &lhs)
2 {
3     IVector &result = *rhs.clone();
4     result += lhs;
5     return &result;
6 }
```

Эта идиома крайне полезна и возмещает очевидный недостаток языка C++ в части виртуального конструирования.

**Вопрос к студентам:** но почему такой недостаток вообще есть? Что мешает сделать в языке настоящие виртуальные конструкторы?

## 3.7 Наследование реализации

*Inheritance is the base class of Evil*

– Sean Parent

Разработанные в (3.6.1) классы-наследники `CelestialBody` и `Planet` всё ещё имеют слишком много дублирующегося кода: общие поля координат и скоростей, общие и притом идентичные селекторы координат. Это обычная и, более того, достаточно частая ситуация. Для уменьшения дублирования в классе `Planet`, можно заметить, что планета также **является** небесным телом, отличаясь только тем, что у неё имеет значение радиус и есть дополнительный метод для определения столкновений. Это отношение “быть частным случаем” также, как и отношение “реализовать интерфейс” моделируется открытым наследованием, но в данном случае унаследована будет и реализация некоторых методов и поля класса `CelestialBody`, такие как `x` и `y`. Общий интерфейс тот же:

```

1 struct ICelestialBody
2 {
3     virtual double get_x () const = 0;
4     virtual double get_y () const = 0;
5     virtual int move (double dt) = 0;
6     virtual int apply_force (double f, double dt) = 0;
7     virtual ~ICelestialBody() {}
8 };

```

Пользовательский тип `CelestialBody` всё так же реализует общий интерфейс:

```

1 class CelestialBody: public ICelestialBody
2 {
3 protected:
4     double x, y;
5     double vx, vy;
6     double m;
7 public:
8     /* ... */
9     /* ICelestialBody implementation */
10    double get_x () const { return x; }
11    double get_y () const { return y; }
12    int move (double dt);
13    int apply_force (double f, double dt);

```

```

14  /* ... */
15 };

```

Новое ключевое слово `protected` сообщает, что все помеченные им поля при открытом наследовании будут доступны производным классам.

Пользовательский тип `Planet` сообщает, что является уточнением `CelestialBody`:

```

1 class Planet: public CelestialBody
2 {
3     double r;
4 public:
5     int detect_collision (const ICelestialBody &rhs) const;
6 };

```

Теперь есть необходимость определить расстояние между небесными телами. Соответствующий обобщённый код:

```

1 double
2 get_distance (const ICelestialBody &lhs,
3                 const ICelestialBody &rhs)
4 {
5     double xdist = lhs.get_x() - rhs.get_x();
6     double ydist = lhs.get_y() - rhs.get_y();
7
8     return sqrt (xdist*xdist + ydist*ydist);
9 }

```

Вполне может быть вызван так:

```

1 Planet Jupiter;
2 Planet Earth;
3
4 /* ... */
5
6 double d = get_distance (Jupiter, Earth);

```

Несмотря на то, что функций `Planet::get_x` и `Planet::get_y` по-просту не существует, у класса `Planet` тем не менее есть эти методы, унаследованные от класса `CelestialBody`. Внутри функции `Planet::get_distance`, динамический тип `Planet` аргумента `lhs` позволяет сделать вызов общей для `Planet` и `CelestialBody` функции, которая изменяет их общую открытую часть.

TODO: где-то здесь про vtable not found problem (случай интерфейса с недоопределёнными функциями).

Существенное отличие открытого наследования от наследования интерфейса – класс не только даёт обещание, что будет поддерживать тот или иной интерфейс, но ищё и заимствует у базового класса конкретную реализацию, оставляя себе только уточнение специфичных для себя методов. Эта техника существенно улучшает переиспользование кода и может быть в произвольной пропорции смешана с чисто абстрактными классами: например вполне можно было бы внести координаты и их селекторы в интерфейсный класс, у которого в этом случае появилась бы часть интерфейса который наследники должны реализовать (чисто виртуальные функции) и часть реализации, которая просто наследуется для использования. Впрочем смешивать такие вещи – не всегда полезно.

Теперь UML-диаграмма становится более иерархичной, как показано на (рис. 3.5).

Можно заметить, что сверху вниз по этой диаграмме, более общие понятия предшествуют более частным. Скажем “конкретное небесное тело” это “небесное тело вообще”, а планета это и то и другое. При открытом наследовании все открытые члены базового класса доступны как открытые члены производного класса, а также работает неявное преобразование типа от производного к базовому классу. Говорят, что открытое наследование **моделирует** отношение “is-a”, при котором объект производного класса является объектом базового класса (в том смысле, в каком планета является небесным телом). Это отношение довольно важно, подробнее см. (3.10.3).

Увы, с точки зрения ООП на C++ такая диаграмма имеет фундаментальный и очевидный недостаток: конкретный класс `CelestialBody` является базовым при наследовании. Это может привести к одной неприятной проблеме.

### 3.7.1 Проблема срезки

Можно ищё раз рассмотреть функцию `detect_collision`, на этот раз остановившись на её параметрах. В диаграмме (рис. 3.5) не очевидно, что там должен стоять указатель на `ICelestialBody`. Кажется достаточно простого указателя на `CelestialBody` – в конце концов это по факту общий интерфейс для всех объектов у которых можно запросить координаты.

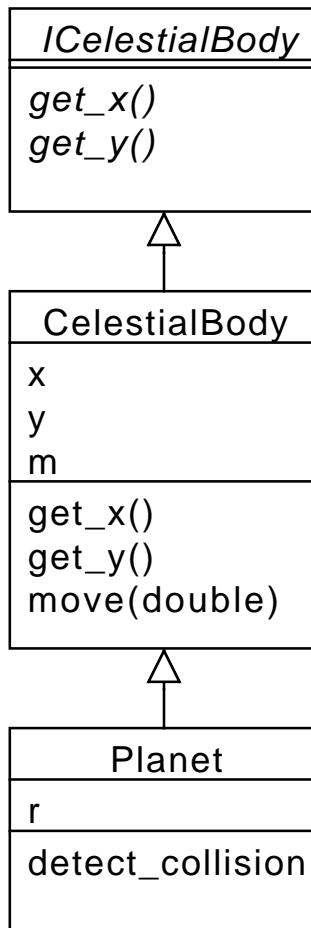


Рис. 3.5: Открытое наследование реализации

Увы, это делает возможным передачу аргументов по значению.

1 `Planet::detect_collision (CelestialBody rhs) /* Oops */`

Проблема понятна: формальные аргументы, переданные по значению конструируются при входе в функцию. Попытка сконструировать объект абстрактного класса (то есть класса, содержащего хотя бы один чисто виртуальный метод) сама по себе ошибочна. Но здесь компилятор уже не защищает от такого ошибочного конструирования. В этом случае новосконструированный объект *rhs* будет иметь динамический тип *CelestialBody* независимо от динамического типа аргументов.

1 `Planet jupiter, earth;`  
2 `jupiter.detect_collision(earth); // sliced`

Здесь переданная планета была срезана до обычного небесного тела, то есть потеряла как минимум поле с радиусом. Для более сложных классов, их состояние вообще может потерять тот или иной инвариант.

В лучшем случае срезки вызов функции `get_x()` будет связан с чисто виртуальной заглушкой интерфейса и программа будет прервана по ошибке.

Этот срез информации при передаче по значению называется “проблемой срезки” (object slicing). Срезка часто возникает в практических контекстах и важно о ней помнить.

### 3.7.2 Идиома NVI и разделение обязанностей

Одним из способов бороться с проблемой срезки и одновременно сохранить наследование реализации может быть вынесение большинства функциональности и даже полей в абстрактный базовый класс, как показано на (рис. 3.6)

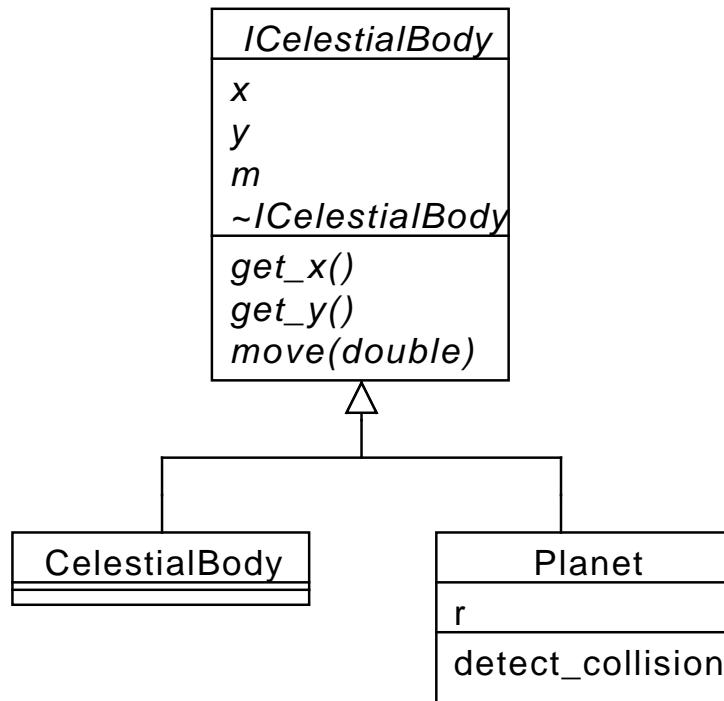


Рис. 3.6: Способ делать конкретными классами только листья

**Домашняя наработка:** напишите код к этой UML-диаграмме

Делать конкретными классами только листья – совет который разделяют многие авторы, для примера можно указать Майерса [5].

Но такой подход порождает проблемы: функция `get_x` больше не требует переопределения в наследниках (все ещё допуская его) и поэтому кто-то может забыть её переопределить, получив при этом странное поведение по умолчанию. Здесь уместен известный анекдот про кенгуру в программе-тренажере для летчиков (класс кенгуру был для простоты и от нехватки времени унаследован от класса пехотинец с соответствующими последствиями во время работы программы – пилоты были поражены тактической слаженностью кенгуру а так же попытками стаи кенгуру запустить по самолету “стингер”).

Выходом является разделить обязанности предоставления поведения по умолчанию и заглушки для переопределения:

```

1 struct ICelestialBody {
2     public:
3         /* Non-Virtual interface */
4         double get_x () const { return do_get_x(); }
5         double get_y () const { return do_get_y(); }
6         virtual ~ICelestialBody() = 0 {}
7
8     protected:
9         double m_x, m_y;
10
11    private:
12        virtual double do_get_x () = 0 { return m_x; }
13        virtual double do_get_y () = 0 { return m_y; }
14    };

```

**Домашняя наработка:** нарисуйте UML-диаграмму с планетами и небесными телами к этому коду.

Эта идиома называется NVI (non-virtual interface) и крайне полезна. Один из таких полезных случаев будет рассмотрен далее, в (3.7.3).

Кроме того, эта идиома окажется на удивление полезной при рассмотрении полиморфных аллокаторов в (5.10.6), когда разговор зайдёт за стандартную библиотеку.

### 3.7.3 Тонкости скрытия имён

При наследовании реализации имена имеют значение. Если некий класс `Matrix` определяет две версии `pow` (для возведения в целую степень, что гораздо проще и в вещественную, что сложнее):

```
1 class Matrix {
2 public:
3     virtual void pow(int x);
4     virtual void pow(double x);
5 };
```

То какой-то его наследник может решить переопределить только одну из них:

```
1 class SparseMatrix : public Matrix
2 {
3 public:
4     void pow (int x);
5 };
```

Увы, это плохая идея. При следующем вызове:

```
1 SparseMatrix t;
2 /* .... */
3 t.pow(5.0);
```

Произойдёт вовсе не вызов `Matrix::pow(double)` а вместо этого неявное преобразование типа и вызов `SparseMatrix::pow(int)`. Чтобы избежать такого поведения можно явно ввести имя базового класса в контекст производного класса:

```
1 class SparseMatrix : public Matrix {
2 public:
3     using Matrix::pow;
4     void pow (int x);
5 };
```

Либо можно воспользоваться идиомой NVI и переопределять закрытые функции с разными именами:

```
1 class Matrix {
2 public:
3     /* Non-virtual interface */
4     void pow(int x) { do_pow_int(x); }
```

```

5   void pow(double x) { do_pow_dbl(x); }
6 private:
7   virtual void do_pow_int(int x);
8   virtual void do_pow_dbl(double x);
9 };

```

Здесь снова разделяется поведение по умолчанию и заглушки для переопределения.

### 3.7.4 Закрытое и защищенное наследование

*The object-oriented version of 'Spaghetti code' is,  
of course, 'Lasagna code'*

— Roberto Waltman

Существенно иначе обстоят дела с закрытым наследованием. При закрытом наследовании, неявного преобразования типа от производного к базовому не задаётся, а все открытые методы базового класса становятся закрытыми методами производного класса, как показано в таблице ниже:

| modifier  | public    | protected | private |
|-----------|-----------|-----------|---------|
| public    | public    | protected | private |
| protected | protected | protected | private |
| private   | private   | private   | private |

Важно понимать, что в эту таблицу сведена внешняя видимость полей и методов класса. Внутренняя видимость и видимость для наследников остаётся прежней.

Говорят, что закрытое наследование моделирует отношение part-of (быть частью). В принципе у закрытого наследования нет почти никакой разницы с композицией:

```

1 struct Engine {
2     int start() const;
3 };

```

Теперь этот двигатель может быть включен в космический корабль посредством композиции

```

1 class SpaceShip_p {
2     Engine e;
3 public:
4     int start (double dt) const { e.start(); }
5 };

```

Или же посредством закрытого наследования.

```

1 struct SpaceShip_p : private Engine {
2     int start (double dt) const { Engine::start(); }
3 };

```

Впрочем одна тонкая разница будет видна при разговоре о множественном наследовании.

Защищенное наследование не слишком популярно в C++, однако автору удалось найти одно интересное применение (вынесено в тест по этой части курса).

### 3.7.5 Конструкторы базовых классов

При конструировании каждого класса будут неявно вызваны конструкторы его базовых классов по умолчанию, если они не вызваны явно в списке инициализации. Если эти конструкторы нетривиальные, то они, соответственно **должны** быть явно вызваны в списке инициализации:

```

1 class A
2 {
3     int x_;
4 public:
5     A(int x) : x_(x) {}
6 };
7
8 class B : public A
9 {
10 public:
11     B() {} // error!
12     B(int x) : A(x) {} // ok
13 };

```

Здесь ошибка при попытке дефолтной инициализации вызвана тем, что подобъект класса A должен быть инициализирован, но нет.

Необходимо понимать, что время жизни объекта начинается после того, как отработал его конструктор. Это означает, что следующий код (хотя и корректен с точки зрения языка) не верен:

```

1 class A {
2     public:
3         A (const char *s) {}
4         const char *f() { return "hello, world"; }
5     };
6
7 class B : public A {
8     public:
9         B() : A ( s = f() ) {}
10    private:
11        const char *s;
12    };

```

Здесь сразу две ошибки. Во-первых попытка вызвать метод `f()` базового подобъекта, который ещё не сконструирован в этой точке (поскольку только вызывается его конструктор). Это может прокатить, может нет, но это в любом случае некорректно. Во-вторых попытка инициализировать и потом использовать ещё не существующий член производного класса (до конструирования которого там пока вообще далеко).

Не следует допускать таких ошибок в своём коде.

### 3.7.6 Больше возможностей для полиморфизма

Начиная с C++11 в язык было введено несколько дополнительных возможностей, которых часто действительно не хватало и которые будут рассмотрены ниже.

В разделе (3.7.3) уже рассматривалась возможность неявных ошибок из-за разрешения имен в стиле C++. Теперь настало время рассмотреть ещё одну ошибку такого рода и что с этим делать. Итак, всё те же матрицы.

```

1 struct Matrix {
2     virtual void pow(int x);
3 };

```

И тут программист совершает обычную человеческую ошибку.

```

1 struct SparseMatrix : Matrix {

```

```

2     void pow (long x); // oops
3 };

```

Увы, эта функцию вовсе не перегружает исходную функцию `pow` и в каком-нибудь сложном динамическом контексте могут быть неприятные сюрпризы.

```

1 Matrix *m = new SpaceMatrix;
2 // ....
3 m->pow(1); // calls Matrix::pow!

```

Действительно, разные сигнатуры засчитываются как разные функции, перегрузки не происходит, всё плохо. Застраховаться от таких ошибок помогает спецификатор `override`.

```

1 struct SparceMatrix : Matrix {
2     void pow (long x) override; // BOOM!
3 };

```

Теперь класс-наследник сообщает компилятору, что он хочет именно переопределять функцию из предка и спасительная ошибка компиляции оповещает, что что-то пошло не так. Это, пожалуй, самое полезное из нововведений 2011-го года и использование его обязательно для всех живых существ, которые не хотят проблем и неприятностей.

Также это свойство позволяет стабильно использовать тонкие места языка. Например при перегрузке функции тип возвращаемого значения (но только его) может быть изменён на обобщающий тип и это корректно.

```

1 template <typename T> struct Base {
2     virtual Base* foo(Base *ptr);
3 };
4
5 template <typename T> struct Derived {
6     Derived* foo (Base *ptr) override; // ok
7 }

```

Но, положа руку на сердце, только ключевое слово `override` делает такую конструкцию не слишком стрёмной: теперь мы на уровне диагностики компилятора уверены, что это корректно. И что следующий код некорректен.

```

1 template <typename T> struct Derived {
2     Derived* foo (Derived *ptr) override; // fail
3 }

```

Таким образом становится возможным вынесение спорных конструкций на суд компилятора.

Ещё один спецификатор, тоже полезный для управления наследованием, вообще запрещает дальнейшее переопределение. И это `final`.

```
1 struct SparceMatrix : Matrix {  
2     void pow (long x) final; // BOOM!  
3 };
```

и одновременно

```
1 struct SparceMatrix : Matrix {  
2     void pow (int x) final; // Ok  
3 };  
4  
5 struct MySparceMatrix : SparceMatrix {  
6     void pow (int x) override; // BOOM!  
7 };
```

Ещё одно ограничение языка – невозможность запретить наследование тоже долго преследовало программистов. Долгое время действовало неформальное правило – нельзя наследоваться если нет виртуального деструктора, но все очень любили его нарушать. К счастью, эту проблему тоже решает модификатор `final`.

```
1 struct Matrix final;  
2 struct SparceMatrix : Matrix; // BOOM!
```

Использование `final` служит непреодолимым барьером в абстракции, поэтому им не следует злоупотреблять, а следует применять только когда он необходим.

## 3.8 Множественное наследование

*The one indisputable fact about MI in C++  
is that it opens up a Pandora's box of complexities  
that simply do not exist under single inheritance*

– Scott Meyers

До сих пор рассматривалось только одиночное одноуровневое наследование, но C++ даёт в этом отношении гораздо больше свободы. Один класс может наследовать многим классам, которые сами кому-то наследуют и так далее. Для этого базовые классы с их модификаторами доступа перечисляются через запятую

```

1 class Man: public AnimalsWithTwoLegs,
2         public OnesWithoutWings
3 {
4     // TODO: God, implement it
5 };

```

Это позволяет строить иерархии взаимодействующих классов и объектов при проектировании сложных программных систем.

Проектирование хорошей иерархии это всегда сложный инженерный процесс, выходящий за рамки этого курса. Зато можно на игрушечных примерах рассмотреть основные проблемы, ожидающие разработчика на этом пути.

### 3.8.1 Виртуальные функции при множественном наследовании

При одиночном наследовании, отношение *is-a*, задаваемое открытым наследованием не вызывает проблем при использовании такой стандартной техники как таблицы виртуальных методов. На (рис. 3.8) изображена условная схема размещения в памяти объекта класса B, определяемого как:

```

1 class A { /* A part */ };
2 class B : public A { /* B part */ };

```

Видно, что указатель `A*` в памяти указывает в точности туда же, куда `B*` и одна и та же таблица виртуальных методов вполне достаточна.

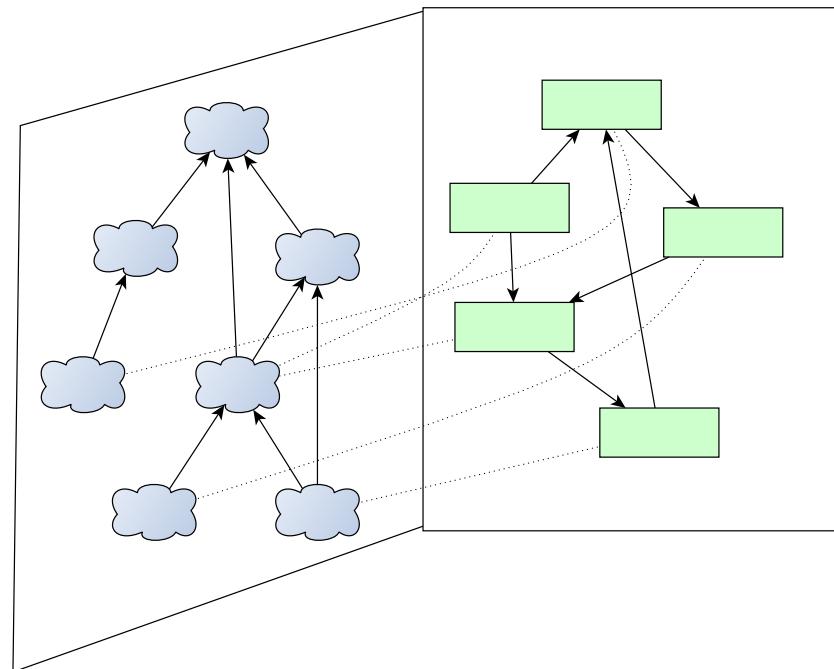


Рис. 3.7: Иерархии классов и объектов

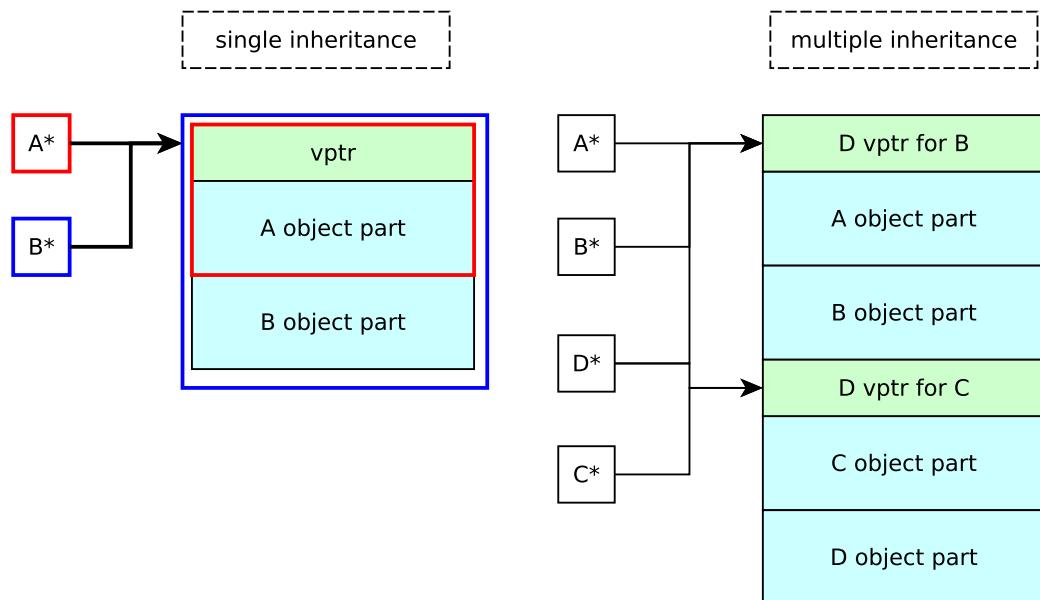


Рис. 3.8: Виртуальные функции при множественном наследовании

Для прогулок по такой иерархии вполне достаточно статического приведения типа:

```

1 B* ptrB = new B;
2 A* ptrA = ptrB;
3 ptrB = static_cast<B*>(ptrA);

```

Увы, для множественного наследования вечер резко перестаёт быть томным. Справа на (рис. 3.8) изображена ситуация, которая получается если добавить пару классов к иерархии:

```

1 class C { /* C part */ };
2 class D : public B, public C { /* D part */ };

```

Даже если предположить, что две таблицы виртуальных функций будут совмещены, всё равно неясно куда должен показывать `C*` если учесть, что в объекте после его разыменования или срезки просто не должно быть никаких частей от классов `A` и `B`, к которым он вообще не имеет отношения.

```

1 D* ptrD = new D;
2 B* ptrB = ptrD;
3 A* ptrA = ptrD;
4 C* ptrC = ptrD; // Where to point?

```

К счастью стандарт регламентирует, что это не ваша головная боль, а головная боль компилятора. Компилятор статически знает, что при множественном наследовании преобразования некоторых типов на добавку постоянного смещения дороже, чем преобразования некоторых других типов (это безумие реализуется в бэкенде GCC через `virtual function thunks` – специальные участки кода, в которых бэкенд решает сколько добавить к указателю при вызове виртуальной функции).

Теперь настало время разобраться с чем-то, что является **вашей** головной болью.

### 3.8.2 Ромбовидные схемы и виртуальные базовые классы

Предположим, вы проектируете систему, поддерживающую абстракции файлов ввода и вывода. Рано или поздно вы пришли к чему-то вроде такого

```

1 class File {};
2 class InputFile : public File {};
3 class OutputFile : public File {};

```

```
4 class IOFile : public InputFile, public OutputFile {};
```

Графически это может быть выражено ромбовидной схемой

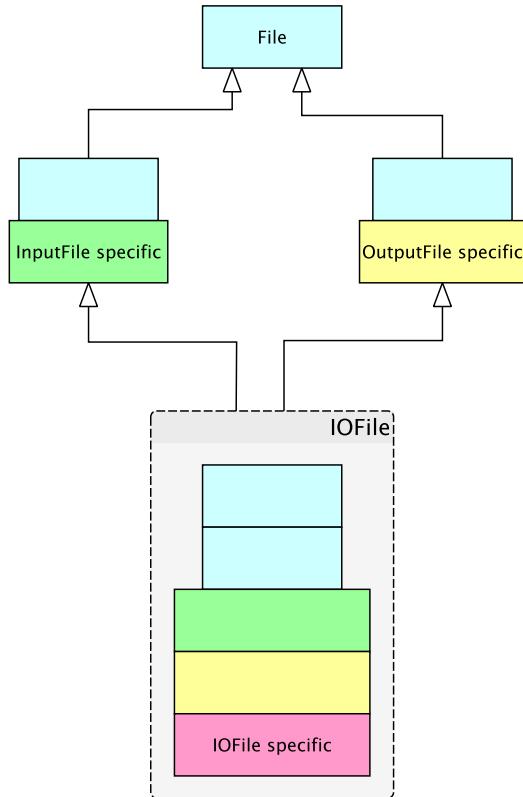


Рис. 3.9: Ромбовидная схема

Предположим, что в классе `File` есть поле `File::filename`. По умолчанию, в классе `IOFile` у вас получится два члена: `InputFile::File::filename` и `OutputFile::File::filename`, но файл у которого два имени это абсурд.

Ещё хуже то, что отношение `is-a` похоже перестаёт работать в таких случаях:

```
1 IOFile* iof = new IOFile;
2 File *f = iof; // Oops!
```

Здесь абсолютно неясно на какую из двух частей `InputFile::File` и `OutputFile::File` теперь указывает `f`. И вот это уже гораздо более серьёзная проблема, чем просто чуть больше оверхеда.

Прошаренный в C++ пацан предложил бы следующий способ устранения неоднозначности (disambiguation).

```
1 File* outPtr = static_cast<OutputFile*>(iof);
2 File* inPtr = static_cast<InputFile*>(iof);
```

Но такие прекрасные хаки никак не спасают от фундаментальных проблем при вызове виртуальных функций. Если переопределение функции `open` предлагается только классами `File` и `OutputFile`, что должно произойти при вызове `inPtr->open?`

Чтобы избежать такой ситуации, базовый класс в наследовании может быть объявлен виртуальным (ещё можно вообще никогда ничего не писать на C++, а использовать C, это предпочтительный вариант).

```
1 class File {};
2 class InputFile : virtual public File {};
3 class OutputFile : virtual public File {};
4 class IOFile : public InputFile, public OutputFile {};
```

Теперь всё хорошо и в ромбовидной схеме у самого нижнего производного класса есть только одна копия базового. В памяти это может выглядеть как-то так:

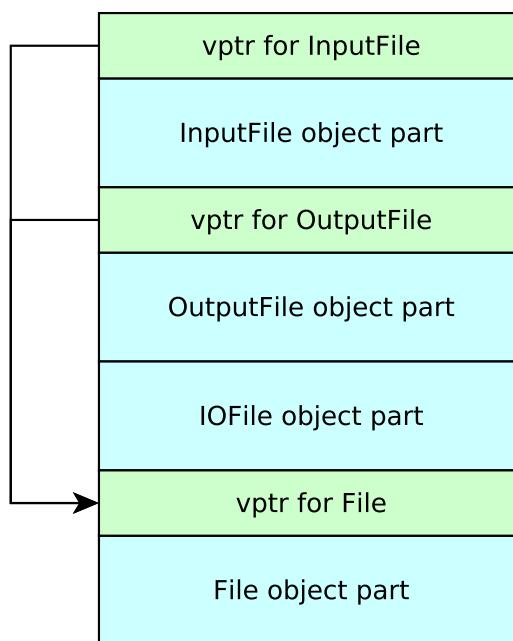


Рис. 3.10: Виртуальное наследование

Таблицы виртуальных методов ещё более усложнились. Показанные стрелки позволяют во время исполнения как бы “склеивать” подобъекты, получая указатели на них.

```
1 File* ptrF1 = ptrInputFile;
2 File* ptrF2 = ptrOutputFile;
3 File* ptrF3 = ptrIOFile;
```

Всё это будет работать и возвращать правильные (по крайней мере – правильно себя ведущие) указатели. Увы, любое обращение по такому указателю связано уже с тремя уровнями косвенности: разрешение виртуального наследования, разрешение множественного наследования и далее собственно вызов по косвенности виртуального метода. Это указатель на указатель на указатель на функцию и, обычно, это высокий барьер для оптимизаций в компиляторе.

Ещё одна проблема – виртуальное наследование отключает понижающие статические преобразования:

```
1 IOFile* ptrIO = static_cast<IOFile*>(ptrF3); // Error
```

**Вопрос к студентам:** почему это происходит?

Для того, чтобы выполнять понижающее преобразование, следует воспользоваться специально введенным для этого в язык динамическим приведением типа.

### 3.8.3 Динамическое приведение и RTTI

Кроме всех операторов приведения, рассмотренных в (2.7.5), существует ещё один, разработанный специально, чтобы приводить статический тип к динамическому типу. Он называется `dynamic_cast`.

Его интересной особенностью является то, что он по-разному работает для указателей и для ссылок.

Для указателей

```
1 Derived* temp = dynamic_cast<Derived*>(base);
```

Пытается привести `temp`, типа `Base*` к типу `Derived*`, где `Base` и `Derived` принадлежат одной и той же иерархии. Если `Derived` является базовым классом для `Base`, то это ничем не отличается от `static_cast`. Но `dynamic_cast` работает также если `Base` является полиморфным базовым классом для `Derived`, то есть является базовым классом для `Derived` и

содержит виртуальные функции. Звучит запутано? Давайте посмотрим пример.

```
1 struct NamedObject {
2     virtual const char *whoareyou() const = 0;
3 };
4
5 struct Pokemon : NamedObject {
6     const char *whoareyou() const { return "Pokemon"; }
7 };
8
9 struct DartVeider : NamedObject {
10    const char *whoareyou() const { return "Dart Veider"; }
11 };
12
13 int check_DartVeider(NamedObject *p)
14 {
15     if (dynamic_cast<DartVeider *>(p))
16     {
17         printf("You are Dart Veider!\n");
18     }
19     else
20     {
21         printf("You are not Dart Veider, you are %s\n", p->
22               whoareyou());
23     }
24
25     return 0;
26 }
```

```
1 DartVeider dv;
2 Pokemon pm;
3 check_DartVeider(&dv);
4 check_DartVeider(&pm);
```

Что будет на выдаче? Ответ:

```
You are Dart Veider!
You are not Dart Veider, you are Pokemon
```

Коротко говоря, `dynamic_cast`, использованный для указателя даёт возможность “спросить” такой ли у этой переменной динамический тип,

как он полагает. Он приводит к этому типу если ответ “да” или возвращает `nullptr`, если ответ “нет”.

При использовании `dynamic_cast` для ссылок, вопрос превращается в утверждение. Если это утверждение нарушается, то выбрасывается исключение и без специальной обработки, программа будет завершена (подробнее про обработку таких ситуаций см. 2.8).

**Вопрос к студентам:** может ли `dynamic_cast` работать везде, где работает `static_cast`?

Именно такой вид преобразования является законным понижающим преобразованием в виртуальных иерархиях.

```
1 IOFile* ptrIO = dynamic_cast<IOFile*>(ptrF3); // Ok
```

Важно понимать: любое использование `dynamic_cast` это легальный способ попросить компилятор сделать вашу программу гораздо медленнее и несколько больше по размеру. Дело в том, что для своей работы он использует запросы к так называемой run time type information или RTTI – обычно это структура, привязанная к виртуальной таблице. Измерить (или хотя бы оценить) точный оверхед обращения к этой структуре и её хранения невозможно. Считается, что если вы используете `dynamic_cast`, вы знаете что делаете.

Из опыта автора этих лекций, обычно люди, использующие его, вообще не знают что делают. Даже примерно.

### 3.8.4 Информация о типах и идентификаторы типов

Каждому типу в C++ соотнесен дескриптор, имеющий тип `type_info`. Этот дескриптор для любого типа можно получить вызовом функции `typeid`. Такие объекты содержащие информацию о динамических типах можно сравнивать. Можно также, начиная с 2011, сравнивать их хеш-коды.

Таким образом, есть два варианта сравнения совпадает ли динамический тип с заданным.

1. Использовать `dynamic cast`

```
1 if (dynamic_cast<A*> someVar != 0) {
2     //it is of class A, or X, inherited from A
3 }
```

2. Использовать typeid

```

1 if (typeid(someVar) == typeid(A)) {
2     //it is of class A
3 }
```

Первый способ позволяет сравнить нечетко, динамический тип может быть приведен к A из любого наследника. Во втором случае сравнение будет четким, а главное – дешевым. Сравнить идентификаторы типов имеет постоянную сложность. Динамическое приведение типов требует обхода деревьев RTTI и может работать непредсказуемо долго.

**Вопрос к студентам:** зачем же вообще нужен `dynamic_cast` если есть `typeid`?

### 3.8.5 Сюрпризы в конструкторах при виртуальном наследовании

Пусть виртуальный базовый класс имеет нетривиальный конструктор как в (3.7.5).

```

1 class A
2 {
3     int x_;
4 public:
5     A(int x) : x_(x) {}
6 };
```

И является виртуальным базовым для классов:

```

1 // C is similar
2 struct B : virtual public A {
3     B(int x):A(x){}
4 };
```

Увы, ожидаемый порядок конструирования для класса наследующего по ромбовидной схеме не сработает интуитивно:

```

1 struct D : public B, public C {
2     D(int x) : B(x), C(x) {} // Error!
3 }
```

Дело в том, что виртуальный базовый класс должен быть обязательно сконструирован в самом нижнем наследнике, то есть:

```
1 D(int x) : A(0), B(x), C(x) {}
```

Мало того, подобъект А будет сконструирован именно с параметром 0, те параметры, которые передаются к общей базе через конструкторы подобъектов В и С, будут проигнорированы.

Это распространяется и ниже по иерархии туда, где об общем виртуальном предке все уже давно забыли:

```
1 struct E : public D {
2     E() : D(10) {} // Error!
3 };
```

Теперь самый нижний класс в иерархии внезапно Е. Это значит, что виртуально-базовая часть должна быть сконструирована в нем.

### 3.8.6 Порядок инициализации в сложных диаграммах

Важно очень хорошо представлять себе порядок конструирования сложных диаграмм классов. Ниже приведен несколько синтетический (но на самом деле гораздо менее сложный, чем многие практически важные иерархии) пример.

```
1 class B1 {};
2 class V1 : public B1 {};
3 class D1 : virtual public V1 {};
4 class B2 {};
5 class B3 {};
6 class V2 : public B1, public B2 {};
7 class D2 : virtual public V2, public B3 {};
8 class M1 {};
9 class M2 {};
10 class X : public D1, public D2 {
11     M1 m1_;
12     M2 m2_;
13 };
```

Для простоты можно изобразить эту иерархию на листочке:

Порядок инициализации объекта класса Х для изображённой иерархии следующий:

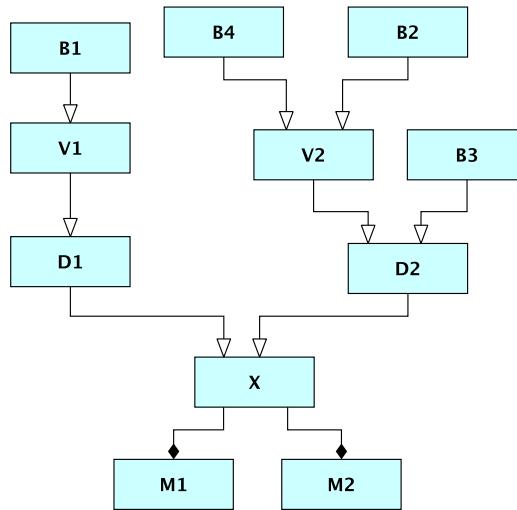


Рис. 3.11: Сложная иерархия

- Сначала конструируются виртуальные базовые классы
  1. Конструирование V1: B1::B1(), V1::V1()
  2. Конструирование V2: B1::B1(), B2::B2(), V2::V2()
- Затем конструируются невиртуальные базовые классы
  1. Конструирование D1: D1::D1()
  2. Конструирование D2: B3::B3(), D2::D2()
- Затем конструируются члены M1::M1(), M2::M2()
- И в последнюю очередь выполняется конструктор X::X()

Вид наследования (открытое, закрытое или защищённое) не влияет на порядок инициализации.

### 3.8.7 Делегация сестринскому типу

Сложности, вносимые множественным наследованием в работу механизма виртуальных функций, частично окупаются теми возможностями, которые они предоставляют.

Пусть задана ромбовидная иерархия, в которой виртуальная функция определена в одной, но не в другой ветке:

```

1 struct VBase {
2     virtual void foo () = 0;
3     virtual void bar () = 0;
4     virtual ~VBase () = 0 {}
5 };
6
7 struct Base1 : virtual public VBase {
8     // do not override bar
9     // but CALL it
10    void foo() override { bar(); }
11 };
12
13 struct Base2 : virtual public VBase {
14     void bar() override {}
15 };
16
17 struct S : Base1, Base2 {}
```

На что может рассчитывать пользователь такого класса, вызывая метод `foo` объекта класса `S`?

```

1 S x;
2 x.foo();
```

Происходит интересная вещь: поскольку метод `foo` не переопределён в классе `Base2`, будет без вариантов вызван метод `Base1::foo`, который внезапно далее вызовет `Base2::bar`. Несмотря на то, что классы `Base1` и `Base2` находятся на одном уровне, механизм виртуальных функций позволяет такие братские (ну или сестринские) делегации.

### 3.8.8 Вложенные классы и снова о пространствах имён

Вложенные функции в C++ невозможны так же как и в С (что кстати не вполне логично, так как вложенные лямбда-функции в новом стандарте возможны и будут рассмотрены на следующих лекциях, так что казалось бы гулять так гулять). Зато C++ позволяет вкладывать классы. Синтаксис очевиден, но, если необходимо, чтобы вложенный класс “знал” о своём окружении, об этом надо позаботиться отдельно:

```

1 class DeathStar {
2     DartVeider m_dv;
3     const char *m_name;
```

```
4 public:
5     DeathStar(const char *name) : m_name(name) {
6         m_dv.starptr = this;
7     }
8
9     struct DartVeider {
10         void whoareyou() const{
11             auto where = starptr ? starptr->m_name
12                             : "hmm... I don't know where";
13             std::cout << "I am Dart Veider, flying on "
14                           << where << std::endl;
15         }
16         DeathStar *starptr;
17     };
18
19     void ask_dart_veider() const { m_dv.whoareyou(); }
20 };
```

Используется это как-то так.

```
1 DeathStar d("Old Good Death Star");
2 d.ask_dart_veider();
```

Каждый вложенный класс определяет область видимости своих имён. Это позволяет гибко разграничивать уровень доступа в зависимости от архитектуры системы.

## 3.9 Скажи мне кто твой друг

*C++ : Where friends have access to your private members*

– Gavin Russell Baker

Модель инкапсуляции C++ логична и стройна (общий смех). В общем случае закрытые члены класса формируют его состояние и недоступны извне, иначе, чем через его методы, отражающие поведение – это хорошо и правильно. Но в некоторых случаях, разработчик хотел бы дать некоему классу эксклюзивный доступ к закрытым членам другого класса. Например для того, чтобы не писать кучу геттеров и в то же время сохранить консистентность абстракции. Для этого язык C++ предусматривает механизм друзей (**friend**) класса.

### 3.9.1 Обычная дружба

Наиболее типичным применением дружбы является дружба между классами. Для этого объявление класса, которому хочется открыть внутреннее состояние должно быть предварено ключевым словом **friend**.

```

1 class Node {
2     int data;
3     int key;
4     /* ..... */
5
6     /* class BinaryTree can access data directly */
7     friend class BinaryTree;
8 };

```

Класс **Node**, назначив себе друга **BinaryTree**, открыл ему всё свое закрытое состояние, но, при этом, сам не получил никакого доступа к **BinaryTree**. Теперь внутри **BinaryTree** можно написать код вида:

```

1 class BinaryTree {
2     Node *root;
3
4     public:
5         int find(int key);
6     };
7
8     int BinaryTree::find(int key) {

```

```

9  // check root
10 if(root->key == key) {
11     // no need to go through an accessor function
12     return root->data;
13 }
14 // rest of find
15 }
```

Класс может открывать своё состояние не только классам, но и функциям. Скажем, не открывая доступ всему `BinaryTree`, можно открыть его только методу `find`:

```

1 class Node {
2     int data;
3     int key;
4     // .....
5
6     // BinaryTree::find can access data directly
7     friend int BinaryTree::find();
8 };
```

Открыть внутреннее состояние можно и отдельной функции. Как дополнительный плюс – в этом случае функция считается объявленной.

```

1 struct Node {
2     // declaration
3     friend int find();
4 };
5
6
7 // find declared and can be used
8 int t = find();
9
10 // definition
11 int find() {
12     /* .... */
13 }
```

Благодаря такому свойству объявлений, дружба может быть полезна во многих неожиданных контекстах при работе с шаблонами (см. 4.3.6).

### 3.9.2 Адвокат дружбе не помеха

К сожалению, в языке не реализовано полноценное открытие части состояния. Семантика дружбы диктует “все или ничего”. Но редко когда нужно, чтобы друзья имели доступ ко всему подряд. Пусть рассмотренный выше класс `Node` согласен открыть классу `BinaryTree` только свои поля `data` и `key`, оставив все остальные закрытыми. В этом случае классу `Node` следует прибегнуть к услугам адвоката (английское слово `attorney` означает скорее “проверенный”).

Игра слов даёт название идиоме `attorney-client` (адвокат-клиент). Адвокат, которым мог бы обзавестить `Node` может выглядеть вот так

```

1 class NodeAttorney {
2 private:
3     static int& data(Node &c) {
4         return c.data;
5     }
6     static int& key(Node &c) {
7         return c.key;
8     }
9     friend class BinaryTree;
10};

```

Теперь внутри `Node` можно оставить только объявление дружбы со своим адвокатом, прочие же объявления дружбы вынести в класс, выполняющий роль `attorney`.

Особо параноидальный дизайн иерархии классов может предоставлять много адвокатов на каждое закрытое поле популярного клиента, каждого со своим списком пользователей.

Нужно очень хорошо понимать, что все эти игры свидетельствуют о наличии некоей проблемы между креслом и компьютером.

### 3.9.3 Дружба в иерархиях классов

Дружба не наследуется, но друг статического родительского типа достаточночен, чтобы делать дружественные вызовы полиморфных наследников.

```

1 class Parent
2 {

```

```

3     friend class Family;
4     protected:
5         virtual void Answer() = 0;
6     };

1 class Child : public Parent
2 {
3     private:
4         void Answer() { printf("Child!\n"); }
5 };

1 class Family
2 {
3     friend class Parent;
4     public:
5         void ParentAnswer(Parent *p) { p->Answer(); } // OK
6         void ChildAnswer(Child *c) { c->Answer(); } // FAIL
7     };

1 Child c;
2 Family f;
3 f.ParentAnswer(&c); /* ok */

```

В общем случае, дружба статических типов вредна для инкапсуляции. Но дружба, позволяющая полиморфные вызовы по большой и открытой иерархии это полезно и иногда необходимо.

### 3.9.4 Дружба и виртуальные вызовы

Часто хочется, чтобы функция-друг была виртуальной. К сожалению это невозможно, так как в таблице виртуальных методов на друзей нет места, к тому же у друзей нет неявного полиморфного аргумента, так что они не могут быть полиморфными функциями.

Предположим, что функция `find` должна уметь работать как с `Node` так и с его потомками.

```

1 class Node
2 {
3     public:
4         friend int find(Node& n);
5     };

```

```

6
7 class SparceNode : public Node
8 {
9 public:
10     friend int find(Node& n);
11 };
12
13 SparceNode t;
14 find(t); // oops, we shall use
15         // another approach here

```

Классическим выходом из положения (чем-то напоминающим идиому NVI) является реализация такой функции через закрытую действитель-но виртуальную функцию.

```

1 class Node
2 {
3 protected:
4     virtual int find();
5 public:
6     friend int find(Node& n);
7 };
8
9 class SparceNode : public Node
10 {
11 protected:
12     virtual int find();
13 public:
14     friend int find(Node& n);
15 };
16
17 int find(Node& n) {
18     return n.find(); // virtual call
19 }
20
21 SparceNode t;
22 find(t); // ok

```

Идея здесь в том, что каждый класс знает как лучше искать свои объекты, а общая функция друг выполняет роль прозрачного интерфейса.

## 3.10 Основные принципы ООП

*Design and programming are human activities;  
forget that and all is lost*

– Bjarne Stroustrup

При использовании объектно ориентированного программирования, на практике оказывается полезным придерживаться некоторых принципов.

### 3.10.1 Принцип единственной обязанности

Не только классы и объекты занимаются инкапсуляцией данных. На языке С, данные могут быть инкапсулированы в модуль (объявлены в нём статическими переменными и функциями) и программист, использующий такой модуль, будет оперировать только его открытым интерфейсом. Данные могут быть инкапсулированы и в обычных функциях, когда локальные переменные внутри функции безопасны относительно изменений извне. Общее понятие, определяющее совокупность “своих” данных, используемых для неких “своих” целей называется **контекст**.

Принцип единственной обязанности гласит, что для каждого контекста следует выделять единственную обязанность и полностью инкапсулировать всё, относящееся к деталям её выполнения.

Пример плохого проектирования:

```
1 class IEmail {
2 public:
3     virtual void setHeader (string smtp_header) = 0;
4     virtual void setContent (string plaintext_content) = 0;
5     virtual void send (string Receivers) = 0;
6 };
```

Здесь у класса три области ответственности: он определяет протокол, контент и отсылку письма. Гораздо лучше дать возможность использовать разные виды протоколов и разные виды контента

```
1 class IContent;
2 class IHeader;
3
4 class IEmail {
```

```

5 public:
6     virtual void setHeader (IHeader *header) = 0;
7     virtual void setContent (IContent *content) = 0;
8     virtual void send (string Receivers) = 0;
9 };

```

Основная интенция принципа единственной обязанности: у каждого класса должна быть только одна причина для изменения. На этапе проектирования этот принцип помогает думать о возможных расширениях проекта и правильно разбивать проект на легко модифицируемые взаимодополняющие классы.

### 3.10.2 Принцип открытости и закрытости

В своей простейшей формулировке он звучит так: “каждый контекст должен быть открыт для расширения, но закрыт для изменения”. Это означает, что при наличии контекста, корректно выполняющего одну обязанность, разработка контекста, выполняющего обязанность, которая расширяет данную, исходный код первого контекста должен быть переиспользован, но не изменен. Это особенно важно в производственной среде, когда изменения в исходном коде потребуют проведение пересмотра кода, модульного тестирования и других подобных процедур, чтобы получить право на использования его в программном продукте. Код, подчиняющийся данному принципу, не изменяется при расширении и поэтому не требует таких трудозатрат.

Пример нарушения:

```

1 class IScreen {
2     void drawRectangle ();
3     void drawCircle ();
4     void drawLine ();
5 public:
6     enum shape {RECTANGLE, CIRCLE, LINE};
7 protected:
8     shape m_shape;
9 public:
10    void draw (shape x);
11 };

```

Этот контекст в абсолютной степени открыт для изменения (оно мало того необходимо при попытке добавить любую функциональность), но

закрыт для расширения. Должно быть наоборот:

```

1 class IFigure;
2 class IRectangle: public IFigure;
3 // .... etc
4 class IScreen {
5 public:
6     void draw (IFigure *f);
7 };

```

Теперь при необходимости расширения, достаточно унаследовать новый класс от общего интерфейса.

### 3.10.3 Принцип подстановки Лисков

И реализация интерфейса и открытое наследование реализации это на самом деле одно и то же фундаментальное отношение, которое в англоязычной литературе называется “is-a”, а по-русски – специализацией (не стоит путать с шаблонной специализацией, это очень разные вещи, здесь термин употреблен в логическом смысле как сужение объёма понятия). Есть некая терминологическая несогласованность в том, как называть классы выше и ниже по иерархии наследования. Многие из предлагаемых в современной литературе вариантов, “родитель” и “наследник”, или, скажем, “суперкласс” и “подкласс”, кажутся вводящими в заблуждение. Далее классы, стоящие выше по иерархии, будут называться **базовыми** для классов, стоящих ниже, а классы стоящие ниже в иерархии – **производными** от стоящих выше.

Отношение обобщения и специализации в правильно спроектированной иерархии классов должно подчиняться “принципу подстановки”.

Этот принцип, сформулированный Барбарой Лисков в [26], гласит, что любое свойство объектов базового класса должно быть верно и для объектов производного класса. Иначе говоря, любая функция, работающая с объектом базового класса, должна работать и с любым объектом производного класса. Поскольку удовлетворяющий LSP наследник может быть **подставлен** в контекст, где использован базовый класс, этот принцип называется принципом подстановки.

Самый простой пример нарушения принципа подстановки – перегрузка виртуальных функций базового класса невиртуальными функциями наследника. Более сложные примеры его применения будут рассмотрены в (3.10.7).

### 3.10.4 Принцип разделения интерфейса

Самая простая его формулировка была дана Робертом Мартином в [27] и звучит так: “Клиенты не должны зависеть от методов, которые они не используют”

Принцип разделения интерфейсов говорит о том, что слишком “толстые” интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

Пример нарушения:

```

1 class IWorker {
2     virtual public void work() = 0;
3     virtual public void eat() = 0;
4 };
5
6 class Manager {
7     IWorker *m_w;
8 public:
9     Manager(IWorker *w):m_w(w){}
10    void manage() {
11        m_w->work();
12    }
13 };

```

В этом исключительно сложном, но жизненном примере менеджер реально пользуется только способностью рабочего работать. Но он зависит от интерфейса в котором также включена не используемая им возможность есть. Скажем теперь разработчик решил ввести `class Robot: public Worker`. Что делать с абсолютно лишним для робота интерфейсом? Ну скажем можно установить ланч длиной в одну секунду (плохая идея). Гораздо лучше:

```

1 class IWorkable {
2     virtual public void work() = 0;
3 };
4
5 class IFeedable {
6     virtual public void eat() = 0;
7 };

```

```
8
9 class IWorker : public IWorkable, public IFeedable;
10
11 class Manager {
12     IWorkable *m_w;
13 // .... etc
14 };
```

Здесь интерфейс разделен и класс менеджера больше не зависит от не используемых им особенностей рабочей силы

### 3.10.5 Принцип инверсии зависимостей

Обычный способ думать о зависимостях между модулями – это рассмотрение зависимостей высокогоуровневых модулей от низкоуровневых. Принцип инверсии разворачивает зависимость и гласит, что:

1. Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

### 3.10.6 Закон Деметры

Перечисленные выше пять принципов входят в пятёрку SOLID, предложенную Робертом Маттином. Кроме SOLID-принципов, известны другие, в частности Закон Деметры, который гласит: “Объект А не должен иметь возможность получить непосредственный доступ к объекту С, если у объекта А есть доступ к объекту В и у объекта В есть доступ к объекту С”. Чтобы понять его совсем просто, можно сказать “Наездник должен управлять лошадью, а не ногами лошади”.

Преимуществами закона Деметры является то, что код, разработанный с соблюдением данного закона, делает написание тестов более простым, а разработанное программное обеспечение менее сложно при поддержке и имеет большие возможности повторного использования кода. Так как объекты являются менее зависимыми от внутренней структуры других объектов, контейнеры объектов могут быть изменены без модификации клиентов.

### 3.10.7 Проблемы, возникающие при проектировании открытого наследования

Принцип подстановки Лисков требует навыка при работе с ним. Предположим, вы проектируете иерархию геометрических объектов и столкнулись с необходимостью расположить в ней такие абстракции как “квадрат” и “прямоугольник”. Первой мыслью может быть унаследовать прямоугольник от квадрата – в конце концов прямоугольник вводит ещё одно поле и может быть какие-нибудь дополнительные методы

```

1 class Square {
2 public:
3     Square(double x) : m_x(x) {}
4     virtual ~Square() {}
5     double a() const { return m_x; }
6     virtual double get_sq() const { return a() * a(); }
7 protected:
8     double m_x;
9 };
10
11 class Rect: public Square {
12 public:
13     Rect(double x, double y) : Square(x), m_y(y) {}
14     int b() const { return m_y; }
15     double get_sq() const override { return m_x * m_y; }
16 private:
17     double m_y;
18 };

```

Но такой подход создаёт проблемы, связанные с тем, что прямоугольник не является частным случаем квадрата и здесь нарушается принцип подстановки.

**Домашняя наработка:** опишите проблемы которые может вызвать такое проектирование. Например рассмотрите реализованную в `Square` (и даже виртуальную) функцию `void increase(int times)`, в `times` раз увеличивающую площадь квадрата. Как вы реализуете её для прямоугольника?

## 3.11 Паттерны объектно-ориентированного программирования

*If you want to sell a cat to a computer scientist  
you have to tell him it's object-oriented*

– Roger King

Введенные в 1995 году в книге [28], паттерны проектирования получили заслуженную популярность в инженерной среде как простой и впечатляющий способ говорить об архитектуре на понятном для посвященных языке.

Авторы предлагали классифицировать паттерны по двум критериям: цели и уровню. По цели паттерны делятся на порождающие, структурные и поведенческие, а по уровню – на уровня класса и уровня объекта. Но классификация по уровню на самом деле довольно условна (особенно в языках с развитым метaproграммированием), поэтому ниже паттерны классифицированы по цели.

### 3.11.1 Порождающие паттерны

Наиболее популярные из порождающих паттернов – Фабричный метод и Абстрактная фабрика по сути уже были рассмотрены. Это (с разных сторон) идиома виртуального конструктора в C++ (см. 3.6.7). Паттерн Prototype это виртуальный копирующий конструктор и его рассмотрение оставлено на самостоятельную работу в рамках домашнего задания.

Ещё один паттерн из числа порождающих – печально известный синглтон (Singleton). Увы, это скорее антипаттерн, так как нет почти никаких отличий синглтонов от глобальных переменных.

Поэтому здесь будет рассмотрен менее известный и одиозный, но прекрасно применимый паттерн Builder. Он находит применение во всех случаях, когда необходимо создавать сложно параметризованные объекты. Например в кодогенераторе компилятора, программист может захотеть создавать инструкцию по мнемонике и аргументам. Проблема в том, что конкретное соответствие аргументов с мнемоникой это деталь реализации бэкенда. Так например в конкретной архитектуре могут быть сложение с тремя или четырьмя аргументами, логические инструкции

с двумя или тремя аргументами или даже некие мнемоники с большим количеством аргументов и странным смыслом.

Желаемый синтаксис, который даёт максимальную гибкость, можно представить следующим образом.

```
1 // we want ADD R0, R1, 1
2 MachineInstrBuilder NewMIB(ADD);
3 NewMIB.addReg(R0);
4 NewMIB.addReg(R1);
5 NewMIB.addImm(1);
6 MachineInstr *NewMI = NewMIB.get();
```

Здесь `MachineInstrBuilder` это класс-строитель, который принимает в конструкторе мнемонику и дальше через серию своих методов, таких как `addReg`, `addImm`, `addLabelRef`, `addMem` и так далее позволяет создавать сколь угодно причудливые варианты машинных инструкций. В конце метод `get` может проверять существует ли на самом деле в описании архитектуры созданная инструкция и возвращать `nullptr` (или делать нечто более сложное), если нет.

### 3.11.2 Структурные паттерны

Паттерны Адаптер, Декоратор, Приспособленец и Фасад понятны из названия, не отличаются разнообразием и обычно свидетельствуют о плохой архитектуре.

В качестве интересного структурного паттерна можно рассмотреть паттерн Мост. Этот паттерн позволяет протянуть связь между абстракцией и интерфейсом реализации. При этом тем же мостом могут пользоваться уточненные абстракции, создавая для себя улучшенных реализаций.

Пусть речь идёт о планировании (*scheduling*) исполняемого кода – одной из последних фаз оптимизатора. Каждый планировщик работает по своему алгоритму: жадному или основанному на расчистке критического пути и так далее. Кроме того, планировщик может быть локальным или глобальным. Паттерн мост позволяет развязать друг с другом конкретный интерфейс и конкретную реализацию, как это показано на (рис. 3.12).

Благодаря абстрактному интерфейсу скрывающему реализацию, в этом примере также выполняется ISP.

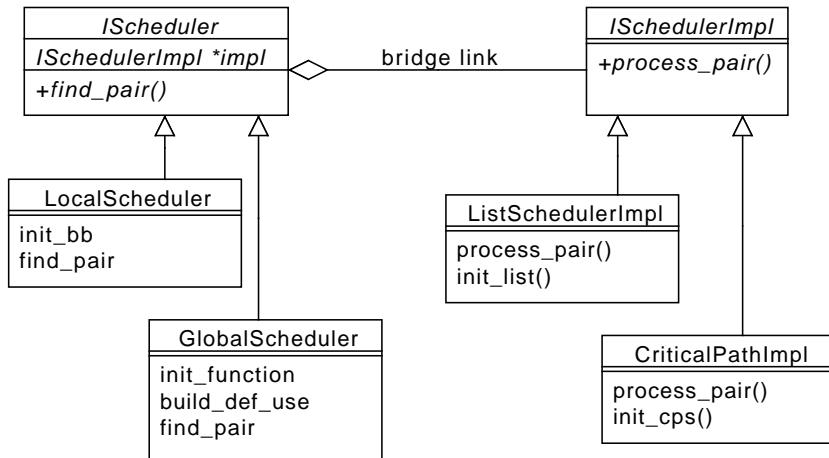


Рис. 3.12: Паттерн Bridge

### 3.11.3 Паттерны поведения

Паттерны поведения – самый интересный раздел, среди них интересны почти все. Например при рассмотрении стандартной библиотеки, будет рассмотрено использование паттерна Итератор (для итераторов) и паттерна Стратегия (для аллокаторов). В качестве характерного примера поведенческих паттернов здесь можно рассмотреть паттерн Наблюдатель.

Этот паттерн позволяет как бы “подписать” один объект на уведомления другого. Пусть объект, который содержит данные будет простым и не содержит ничего кроме целого числа (и вектора наблюдателей).

```

1  class Subject {
2      vector < class Observer * > views;
3      int value;
4  public:
5      void attach(Observer *obs) {
6          views.push_back(obs);
7      }
8      void setVal(int val) {
9          value = val;
10         notify();
11     }
12     int getVal() {
13         return value;
14     }

```

```

15     void notify();
16 };

```

В функции `notify` также нет ничего сложного: она просто уведомляет всех наблюдателей о смене состояния.

```

1 void Subject::notify() {
2     for (int i = 0; i < views.size(); i++)
3         views[i]->update();
4 }

```

Теперь нужно договориться что наблюдают наблюдатели. Предположим, что частное или остаток от деления числа субъекта на заданное число (в реальных программах все будет сложнее, но это учебный пример).

Общий класс задает виртуальный метод `update` для реализации в подклассах.

```

1 class Observer {
2     Subject *model;
3     int denom;
4 public:
5     Observer(Subject *mod, int div) {
6         model = mod;
7         denom = div;
8         model->attach(this);
9     }
10    virtual void update() = 0;
11 protected:
12    Subject *getSubject() {
13        return model;
14    }
15    int getDivisor() {
16        return denom;
17    }
18 };

```

Теперь два подкласса.

```

1 class DivObserver: public Observer {
2 public:
3     DivObserver(Subject *mod, int div): Observer(mod, div){}
4     void update() {

```

```
5     int v = getSubject()->getVal(), d = getDivisor();
6     cout << v << " div " << d << " is " << v / d << '\n';
7 }
8 };
9
10 class ModObserver: public Observer {
11 public:
12     ModObserver(Subject *mod, int div): Observer(mod, div){}
13     void update() {
14         int v = getSubject()->getVal(), d = getDivisor();
15         cout << v << " mod " << d << " is " << v % d << '\n';
16     }
17 };
```

Все это складывается вместе вот в такую замечательную городушку.

```
1 int main() {
2     Subject subj;
3     DivObserver divObs1(&subj, 4);
4     DivObserver divObs2(&subj, 3);
5     ModObserver modObs3(&subj, 3);
6     subj.setVal(14);
7 }
```

**Вопрос к студентам:** и что будет на экране?

## 3.12 Домашняя наработка по ООП

### Контрольные вопросы

1. Может ли нестатический метод в структуре вызывать статический метод?
2. Можно ли создать объект класса в конструкторе этого класса?
3. Можно ли создать объект класса в деструкторе этого класса?
4. Имеет ли смысл ключевое слово `explicit` для конструктора без аргументов?
5. Достаточно ли объявление класса для использования указателей на его методы?
6. Почему у конструкторов и деструкторов нет возвращаемых значений?
7. Можно ли перегрузить деструктор (так же как мы перегружаем конструкторы)?
8. Назовите оба способа задать неявное преобразование типов для вида класса.
9. В лекциях рассмотрены `const` методы. Бывают ли `volatile` методы классов?
10. Может ли возникнуть необходимость объявить поле одновременно `const` и `mutable`?
11. Чем отличается указатель на статический метод от указателя на нестатический?
12. Что означает сокращение RAI, зачем нужна эта идиома?
13. Чем плоха идея реализовать копирующий конструктор через вызов оператора присваивания к `*this`?
14. Может ли быть скопирован по умолчанию объект класса с полем-ссылкой?
15. Что означает сокращение RVO, для чего используется?

16. Чем плоха идея перегрузить фигурные скобки (наравне с квадратными и круглыми)?
17. В лекциях рассматриваются операторы new и sizeof. Почему это операторы, а не функции?
18. Сохраняется ли правая ассоциативность при переопределении += или -=? Почему да или почему нет?
19. Возможно ли переопределить правый декремент для классов с запрещенным копированием?
20. Перечислите стандартные и пользовательские формы new и delete
21. В каких случаях деструктор нужно вызывать явно?
22. Можно ли вызвать (тем или иным способом) конструктор класса?
23. Как выбрать между правой и константной левой ссылкой?
24. Есть ли смысл в константных правых ссылках?
25. В лекциях рассмотрено связывание левой ссылки с правой (класс RefBind). Возможно ли связывание в обратном направлении?
26. Как выбрать между конструктором копирования и перемещения?
27. Зачем в языке и std::move и std::forward если оба это просто static cast?
28. За счет каких механизмов языка мы можем использовать универсальные ссылки?
29. Работает ли std::forward для идеального проброса правых ссылок?
30. Как привести указатель на объект производного класса к ссылке на объект базового?
31. Во сколько обычных вызовов по косвенности может обойтись вызов виртуальной функции?
32. Можно ли получить указатель на чисто виртуальную функцию?
33. В C++ нет статических деструкторов. Если бы они были, что мог бы делать виртуальный статический деструктор?

34. Может ли виртуальная функция быть статической? А чисто виртуальная?
35. Зачем нужны виртуальные деструкторы?
36. Может ли виртуальная функция быть объявлена `inline`?
37. Возможно ли корректно скопировать производный класс по указателю на базовый?
38. Что такое проблема срезки и как она проявляется?
39. Что обозначает аббревиатура `NVI`, зачем нужна эта идиома?
40. Могут ли в чисто виртуальных функциях быть аргументы по умолчанию?
41. В чем отличия композиции от закрытого наследования?
42. Чем отличается виртуальное наследование от невиртуального?
43. Имеют ли смысл виртуальные базовые классы при одиночном наследовании?
44. Почему виртуальное наследование отключает понижающий `static cast` по иерархии?
45. У вас есть выбор между использованием `dynamic cast` и `typeid` для проверки типа, в каком случае чем вы воспользуетесь?
46. Расшифруйте аббревиатуру `SOLID` и объясните кратко каждую из входящих в неё аббревиатур.
47. Что такое закон Деметры?
48. Выберите любой из классических паттернов проектирования и объясните его применимость.

### Задания

1. Расширьте код игры в мяч до трёхмерной
2. Введите в код игры в мяч примитивы синхронизации чтобы его можно было запускать в многопоточном окружении

3. Расширьте код до игры в мяч со смещающимся центром тяжести (допустим мяч наполовину заполнен водой или песком)
4. Разработайте конкретный класс `CTime`, предназначенный для хранения текущего времени и вычисления временных интервалов. Не ограничивайтесь unix time, сделайте возможным, например, подсчёт количества часов от Куликовской битвы до Бородинского сражения
5. Как можно улучшить рассмотренный в (3.2.1) класс `CFile`? Реализуйте и протестируйте обёртку.
6. Разработайте и реализуйте иерархию классов, для управления сетевым соединением. Предусмотрите как минимум классы для TCP и UDP сокетов, наследующие от общего предка
7. Реализуйте абстрактный интерфейс материальной точки и унаследуйте от него разные варианты мяча – идеальный, с сопротивлением воздуха, со смещённым центром тяжести, etc. На основе общего интерфейса, реализуйте игрока в мяч, которому всё равно каким мячом играть
8. Перегрузите для класса комплексных чисел, приведённых в (3.3) инкремент, логические операции, вычитание
9. Разработайте класс кватернионов со всеми перегруженными арифметическими операциями. Исследуйте есть ли смысл наследовать его от комплексных чисел.
10. Разработайте класс для операций над матрицами. Проиллюстрируйте к каким проблемам может привести ставшее некоммутативным умножение.
11. Для вашего класса операций над матрицами перегрузите выделение и освобождение памяти. Разработайте стратегию выделения памяти для разреженных матриц. Проиллюстрируйте умножение двух разреженных матриц 10000x10000 в каждой из которых значимыми являются всего 2-3 элемента (остальные нули).

## Глава 4

# Особая шаблонная магия

*In truth, the only restrictions on our capacity  
to astonish ourselves and each other  
are imposed by our own minds*

– David Blaine

Шаблоны, введённые в C++ были в своё время введены туда экспериментально. Их не было ни в одном другом языке и никто по настоящему не знал, что из этого получится. Поэтому большинство интересных и важных свойств шаблонов не были разработаны. Они были открыты. В этом разделе курса, вас тоже ожидают открытия.

Многие считают шаблонную подсистему языка C++ самой развитой и сложной в мире современного программирования (по крайней мере – мейнстримного). Эта точка зрения небезосновательна. Но шаблонная подсистема не только сложна сама по себе. Ещё интереснее взаимодействие этой подсистемы с объектно-ориентированными возможностями языка, а также потрясающие средства (такие как вариабельные шаблоны, кортежи и лямбда-выражения) которые становятся доступны для улучшения абстракций и порождения более компактного и быстрого кода. Но даже простые шаблоны функций, с которых и начинается изложение, таят в себе массу сюрпризов.

## 4.1 Шаблоны функций

Общий обзор шаблонных функций уже был проведен в (2.7.3), теперь настало время углубиться в детали и обсудить шаблонные функции в подробностях, прежде чем переходить к более сложным шаблонам классов.

**Вопрос к студентам:** как написать пару `min` и `max`?

Возможный вариант ответа использует константные ссылки, но это могли бы быть и значения и правые ссылки.

```

1 template <class T> const T&
2 max (const T &x, const T &y)
3 {
4     return ((x > y) ? x : y);
5 }
6
7 template <class T> const T&
8 min (const T &x, const T &y)
9 {
10    return ((x < y) ? x : y);
11 }
```

Классическим тестом на минимакс является переход

$$\forall x, y | x \leq y \quad \min(x, y) = x \wedge \max(x, y) = y$$

Обобщённые функции можно вызывать из обобщённых функций, так что тест тоже может быть шаблонным:

```

1 template <typename T> bool
2 test_minmax (const T &x, const T &y)
3 {
4     assert (x <= y);
5     return (min<T> (x, y) == x) && (max<T> (x, y) == y);
6 }
```

В данном случае можно протестировать на паре чисел, паре символов, паре чисел с плавающей точкой, etc:

```

1 assert(test_minmax<char> ('a', 'b'));
2 assert(test_minmax<int> (5, 6));
3 assert(test_minmax<double> (3.0, 7.2));
4 /* ... */
```

### 4.1.1 Вывод типов шаблонами

Если бы всегда при работе с шаблонами возникала необходимость явно указывать шаблонные аргументы, код довольно скоро разрастался бы, обрастаю ненужными подробностями. Скажем, так ли нужно писать `test_minmax<int>(5, 6)` если компилятор достаточно умен, чтобы понять и более простую форму `test_minmax(5, 6)` из контекста? Таким образом можно переписать:

```
1 assert(test_minmax ('a', 'b'));
2 assert(test_minmax (5, 6));
3 assert(test_minmax (3.0, 7.2));
4 /* ... */
```

Положившись на вывод типов шаблонными функциями. Он похож на вывод в `auto` (2.11.1), но исторически он появился раньше и работает несколько слабее. Можно рассмотреть шаблонную функцию, принимающую указатель на некую функцию произвольного типа.

```
1 template <typename Func> Func
2 deduce(const Func & f)
```

Здесь для `void f(){}`  выведенным типом при вызове `deduce(f)` будет `void(*)()`. Но что если нужно вывести не тип указателя на функцию, а возвращаемый тип функции? Увы, с классическими шаблонами это сделать нелегко: тип выводится из типа аргумента и только.

**Вопрос к студентам:** какие новые возможности даёт здесь использование `decltype`?

При выводе типов шаблоны сворачивают ссылки (левые и правые) так же, как это делает `auto`.

```
1 template <typename T>
2 void foo(T&& t);
```

Пусть теперь `foo` вызвана с аргументом `x` типа `X`, причём `x` является lvalue. Тогда `T` разрешается в `X&`, а реальным типом аргумента `t` будет `X& &&`, то есть `X&&`. Если же `x` является rvalue, то `T` разрешается в `X` и реальным типом аргумента `t` будет `X&&`.

В некоторых случаях, вывод не работает – например не выводится тип возвращаемого значения:

```
1 // SrcT can be deduced, but DstT cannot
2 template <typename DstT, typename SrcT>
```

```

3  inline DstT implicit_cast (SrcT const& x)
4  {
5      return x;
6  }

```

Здесь этот тип необходимо указывать явно.

```
1 double value = implicit_cast<double>(-1);
```

При этом второй тип легко выводится из типа аргумента и его можно не указывать.

#### 4.1.2 Обобщенность и неожиданности

Пока что с функциями `max` и `min` всё было хорошо – все три теста проходят, в чем несложно убедиться. Но обобщённые функции могут принимать переменные любого рода. Пусть имеется очень простая структура, определяющая имя и возраст человека

```

1 struct Person
2 {
3     const char *name;
4     int age;
5     Person (const char *a_name, int an_age) :
6         name (a_name), age(an_age) {}
7 };
8
9 bool
10 operator > (const Person &lhs, const Person &rhs)
11 {
12     return lhs.age > rhs.age;
13 }
14
15 bool
16 operator < (const Person &lhs, const Person &rhs)
17 {
18     return lhs.age < rhs.age;
19 }
20
21 bool
22 operator <= (const Person &lhs, const Person &rhs)
23 {

```

```
24     return !(lhs > rhs);
25 }
26
27 bool
28 operator == (const Person &lhs, const Person &rhs)
29 {
30     return (&lhs == &rhs);
31 }
```

И вот появляются Иван и Данила:

```
1 Person Ivan ("Ivan", 24);
2 Person Danila ("Danila", 24);
3 cout << test_minmax (5, 6) << endl;
4 cout << test_minmax (Ivan, Danila) << endl;
```

**Вопрос к студентам:** что будет на экране?

Такое поведение функций, когда для пары одинаковых объектов они сохраняют их значения и порядок в результирующей паре, называется стабильностью. Можно сделать вывод, что функции `min` и `max` ведут себя **нестабильно**. Проблемы стабильности также возникают при проектировании обобщённых алгоритмов сортировки, бинарного поиска и многих других.

**Вопрос к студентам:** как переопределить нашу пару функций так, чтобы они стали стабильными?

Какая мораль? Разработка обобщенного кода очень сильно отличается от разработки конкретного кода. Нужно серьёзно вникать в детали, которых может вовсе не возникнуть когда вы работаете с заранее определенными данными. Все дальнейшие лекции будут посвящены обобщенному коду.

**Домашняя наработка:** Для трёх элементов, можно написать три порядковых статистики – максимум, медиану и минимум. Попробуйте написать их в терминах уже написанных бинарных `min` и `max`, предполагая, что бинарные `min` и `max` стабильны. Сможете ли вы сделать тернарные варианты также стабильными?

### 4.1.3 Перегрузка шаблонных функций

Правила перегрузки шаблонных функций расширяют правила перегрузки обычных функций, рассматривавшиеся ранее в (2.6.2), но со спецификой обобщённого кода.

Новая и уточнённая последовательность перегрузки теперь выглядит так

1. Идеальное совпадение
2. **Инстанцирование шаблона функции**
3. Стандартные преобразования
4. Пользовательские преобразования
5. Троеточия
6. Ссылочное связывание
7. Списочная инициализация

Но не всё так просто. Шаблоны функций тоже идут в определённом порядке. Разворачивая выделенный новый пункт, можно записать нечто вроде

1. Идеальное совпадение
2. **Инстанцирование шаблона функции**
  - (a) Шаблоны с меньшим количеством шаблонных параметров
  - (b) Шаблоны с наиболее уточнёнными аргументами
  - (c) Шаблоны общего вида
3. Стандартные преобразования
4. Пользовательские преобразования
5. Троеточия
6. Ссылочное связывание
7. Списочная инициализация

Простые случаи действительно просты. Чтобы разобраться в сложных случаях, надо запомнить всего три совсем несложных правила.

**Правило 1.** Точно подходящая функция всегда выигрывает у шаблона.

```

1 // [1]
2 int const&
3 max (int const& a, int const& b)
4 {
5     return a < b ? b : a;
6 }
7
8 // [2]
9 template <typename T> T const&
10 max (T const& a, T const& b)
11 {
12     return a < b ? b : a;
13 }
```

Немного тренировки для применения этого правила:

```

1 max(7, 42); // [1]
2 max(7.0, 42.0); // [2]
3 max<int>(7, 42); // [2]
4 max<>(7, 42); // [2]
5 max('a', 42.7); // [1]
```

**Правило 2.** Более специальный шаблон всегда выигрывает у менее специального.

```

1 template <typename T> void f(T); // [1]
2 template <typename T> void f(T*); // [2]
3 template <typename T> void f(T**); // [3]
4 template <typename T> void f(T***); // [4]
5 template <typename T> void f(T****); // [5]
```

Теперь при следующем вызове

```

1 int ***a;
2 f(a); // --> [4]
3 f<int**>(a); // --> [2]
```

Второй вариант более специален, поэтому там, где он может быть использован, он выигрывает у первого.

**Правило 3.** Меньшее количество параметров выигрывает против большего, если это не противоречит предыдущему правилу.

Сначала основная ветка правила.

```

1 template <typename T1, typename T2> void f( T1, T2 ); // [1]
2 template <typename T> void f( T, T* ); // [2]
3
4 double t, s;
5 f(t, &s); // --> [2]
```

Но это не работает при конфликте с предыдущим правилом.

```

1 template <typename T> void g( T, T );
2 template <typename T1 typename T2> void g( T1, T2* );
3 template <typename T1 typename T2> void g( T1*, T2* );
4
5 g (&t, &s);
```

Здесь показанный вызов просто не скомпилируется, так как тут меньше параметров у более специального шаблона и это семантический конфликт.

**Домашняя наработка:** рассмотрите случай, когда одновременно объявлены одновременно шаблоны для ссылки и константной ссылки

```

1 template <typename T> inline T&
2 max (T& a, T& b) { /* ... */ }
3
4 template <typename T> inline T const&
5 max (T const& a, T const& b) { /* ... */ }
```

Как в этом случае будет разрешаться перегрузка? Приведите примеры. Аналогично для указателя, константного указателя, указателя на константные данные. Если добавить к перегрузке по ссылке перегрузку по значению, создав тем самым неоднозначность, компилятор должен отреагировать ошибкой. Проверьте как отработает эту ситуацию ваш компилятор.

#### 4.1.4 Управление инстанцированием шаблонов функций

Пусть задана довольно простая шаблонная функция с довольно простым использованием – возведение в квадрат чисел разных типов.

```

1 template<class T> T square(T n) {
2     return n * n;
3 }
4
5 int main () {
6     int x = square (3);
7     float y = square (3.0);
8     return 0;
9 }
```

**Вопрос к студентам:** сколько копий этой функции будет в объектном файле и какие они будут?

Вопрос мотивирует какие-то способы исследования реального результата работы компилятора в области подстановки шаблонов. Приведенный в ответе вызов

```
$ objdump -tC square.o
```

Это Linux-специфичный способ посмотреть таблицу символов объектного файла.

Но что будет если функция оказалась сразу в двух единицах трансляции?

```

1 #ifndef MAX_GUARD_
2 #define MAX_GUARD_
3
4 template <typename T> T
5 max (T x, T y)
6 {
7     return (x > y) ? x : y;
8 }
9
10 extern int foo (int x, int y);
11 extern int bar (int x, int y);
12
13 #endif
```

и в обоих функциях `foo` и `bar`, находящихся в модулях `maxuser1.cc` и `maxuser2.cc` происходит вызов `max(x, y)`?

```
$ g++ maxuser1.cc -c
```

```
$ objdump -tC maxuser1.o
w      F .text._Z3maxIiET_S0_S0_

$ g++ maxuser2.cc -c
$ objdump -tC maxuser2.o
w      F .text._Z3maxIiET_S0_S0_
```

Видно, что теперь функция с одинаковым именем есть в обоих модулях.

**Вопрос к студентам:** почему это нарушение ODR?

Итак, их все ещё можно слинковать вместе и линкер произвольно выберет одну из них (благо обе идентичны).

Увы, если из этих модулей нужно собрать библиотеку:

```
ar cruv libmax.a maxuser1.o maxuser2.o
objdump -tC libmax.a
```

то в этой библиотеке окажется две копии функции. Хуже того: при компиляции каждого модуля эта функция должна быть N раз независимо построена и соптимизирована. Очень плохо. Ещё хуже если некий модуль собирался с одними опциями оптимизации, а другой с другими.

Можно заблокировать инстанцирование, поместив в один из модулей (например в maxuser2.cc) явное указание, что шаблон уже был инстанцирован с этими параметрами где-то ещё:

```
1 extern template int max<int> (int, int);
```

Теперь лишний раз функция инстанцирована не будет:

```
$ g++ --std=c++14 maxuser2.cc -c
$ objdump -tC maxuser2.o
g      F .text 0000000000000024 bar(int, int)
          *UND* 0000000000000000 int max<int>(int, int)
g      F .text 0000000000000038 main
          *UND* 0000000000000000 foo(int, int)
```

Но конечно неясно что такого особенного в модуле maxuser2. Для того чтобы решить эту проблему в более общем виде, можно заблокировать инстанцирование с этим типом везде кроме принудительной точки инстанцирования:

В max.hpp:

```
1 extern template int max<int> (int, int);
```

И в новом модуле max.cc

```
1 #include "max.hpp"
2 template int max<int>(int, int);
```

Эта техника, введенная в C++14 позволяет гибко управлять точками инстанцирования, блокируя или вынуждая компилятор инстанцировать те или иные шаблоны. Здесь речь о шаблонах функций, исследование шаблонов классов – более тонкое искусство, но в целом следует тому же принципу.

#### 4.1.5 Два полиморфизма

Что же, к этому моменту вы знаете о шаблонах функций уже достаточно, чтобы вернуться к серьёзным вещам, таким как планеты и космические корабли. Можно наследовать их от общего предка, как это делалось в (3.6.1), но совершенно не задействовать виртуальные функции:

```
1 class CelestialBody
2 {
3     double x, y;
4 public:
5     double get_x () const;
6     double get_y () const;
7     /* ... other ... */
8 };
9
10 class SpaceShip : public CelestialBody
11 {
12     /* ... spaceship-related ... */
13 };
14
15 class Planet : public CelestialBody
16 {
17     /* ... planet-related ... */
18 };
```

И тем не менее сохранить возможность писать довольно абстрактный код, полагающийся только на общий интерфейс объектов (способность сообщить координаты) и отдающий им на откуп детали реализации:

```

1 template <typename T> double
2 get_distance (const T &lhs, const T &rhs)
3 {
4     double xdist = lhs.get_x() - rhs.get_x();
5     double ydist = lhs.get_y() - rhs.get_y();
6     return sqrt (xdist*xdist + ydist*ydist);
7 }
```

Это прекрасное свойство называется полиморфизмом времени компиляции, или статический полиморфизм. Заметьте, никаких больше виртуальных функций, никаких таблиц виртуальных методов, никакого оверхеда времени выполнения (за счёт раздутия кода помещением туда многочисленных экземпляров функции `get_distance`, разумеется).

Разница в том, что класс `ICelestialBody` создавал явный интерфейс, в случае же статического полиморфизма, шаблонная функция накладывает неявные требования к обоим своим инстанцирующим типам, сформулированные точно так же “поддерживать функцию-член `get_x()`” и “поддерживать функцию-член `get_y()`”.

При рассмотрении нового стандарта будут рассмотрены “концепты”, которые наконец-то определяют способ задать неявный интерфейс явно, см. (5.11).

Ещё более важным примером полезности шаблонного полиморфизма является его “утиная” сущность (если нечто плавает как утка и крякает как утка, то это утка). Например, функторы **не являются** взаимозаменяемыми с указателями на функции. Если некая функция ждёт указатель, передать ей функтор даже с той же сигнатурой – нельзя.

Для примера:

```

1 struct Pred
2 {
3     int operator()(int x);
4 };
```

Этот функтор нельзя использовать с явным интерфейсом:

```

1 int filtered_sum(int numbers[], int len, int (*pred)(int))
2 {
```

```

3   int i, a = 0;
4
5   for (i = 0; i != len; ++i)
6     if (pred(numbers[i])) { a += numbers[i]; }
7
8   return a;
9 }
10
11 Pred p;
12 filtered_sum (nums, 10, p); // error;

```

К счастью, при работе с неявными шаблонными интерфейсами, требование к типу быть выполнимым-как-функция позволяет прозрачно передавать и функторы и указатели на функции в качестве такого типа.

```

1 template <typename FuncType>
2 int filtered_sum (int numbers[],
3                     int len, FuncType pred)
4 {
5   /* ... same body ... */
6 }
7
8 Pred p;
9 filtered_sum (nums, 10, p); // ok;

```

**Вопрос к студентам:** что будет при попытке вызвать функцию `max` передав в качестве аргумента `std::complex`?

#### 4.1.6 Когда C++ быстрее, чем C

Благодаря статическому полиморфизму, даже самые простые шаблоны функций оказываются не так просты. Их не стоит недооценивать.

**Вопрос к студентам:** какую сигнатуру вы напишете, если хотите на С написать обобщённую функцию сортировки?

Почему в этой сигнатуре использован `void*`? Потому что в обобщённой функции не известно какого типа элементы будут отсортированы. Кроме того требуется передавать размер элемента и их количество раздельно, что открывает возможности для человеческих ошибок. Но хуже всего то, что компаратор (вызываемый на каждое сравнение элементов) это указатель на функцию. Из-за этого:

- Каждый вызов компаратора имеет штраф на разыменование указателя на функцию – лишний уровень косвенности
- Исключена возможность inline-подстановки сравнения. При этом сравнение обычно сводится к чему-то очень простому (скажем простому “меньше, чем”) и подставить его бывает очень полезно.

Всех этих недостатков лишена реализация в стиле C++, которая может выглядеть примерно так (предполагаем, что для типа `Element` уже есть оператор сравнения на меньше).

```
1 template <typename Element>
2 void qsortpp (Element *base, size_t nmemb);
```

Это лучше, проще, более безопасно относительно типов и это гораздо эффективнее. Платой за это является объём кода, который компилятор теперь генерирует для каждой инстанциации `qsortpp`, но это не всегда дорого.

На самом деле стандарт C++98 регламентирует (C++98 25.3.1.1) даже более изящную сигнатуру для стандартной сортировки:

```
1 template <class RandIt>
2 void sort(RandIt first, RandIt last);
```

Но вся её мощь, красота и обобщённость проявятся позже, когда речь пойдёт о стандартной библиотеке.

**Вопрос к студентам:** что принципиально мешает ввести шаблоны в язык С (даже не вводя туда классы, просто шаблоны функций)?

**Домашняя наработка:** Реализуйте `qsort` в стиле С и в стиле C++

#### 4.1.7 Когда шаблонный полиморфизм уступает динамическому

Использование шаблонного полиморфизма снижает накладные расходы на исполнение и почти всегда предпочтительно. Но есть вещи, которые требуют динамического полиморфизма и оказываются неоправданно сложными если пытаться сделать их на этапе компиляции.

```
1 struct obj {
2     virtual int foo (int) = 0;
3 };
```

```
4
5 class A : public obj;
6 class B : public obj;
7
8 // .... classes C, D, etc ....
9
10 // returns A if config have 'a', B if 'b', and so on
11 obj * getfromconfig (const char *filename);
12
13 int entry (obj *x) {
14     return x->foo();
15 }
16
17 int main () {
18     return entry (getfromconfig("my.xml"))
19 }
```

Этот код представляет собой идиому “виртуального конструктора” (так же известен как фабричный метод).

**Домашняя наработка:** попробуйте переписать его с использованием только статического полиморфизма.

## 4.2 Шаблоны классов и специализация

Допустим, стоит задача спроектировать стек – класс объектов любого типа (но однородных) которые будут туда помещаться и вытаскиваться. Как обычно синтаксис вполне понятен через пример (детали всегда можно прочитать в стандарте).

```

1 template <typename T>
2 class Stack {
3     struct StackElem {
4         T elem;
5         StackElem *next;
6     } *m_top;
7 public:
8     Stack() : m_top(NULL) {}
9     void push(const T& elem);
10    void pop();
11    T top() const { return m_top->elem; }
12 };

```

Обратите внимание на то, что `T`, помещённый в шаблон, используется внутри класса как совершенно обычный тип. Можно вернуть его из метода, передать в метод и даже использовать как деталь реализации скрытой подструктуры.

```

1 template <typename T>
2 void Stack<T>::push(const T& elem) {
3     StackElem *newelem = new StackElem;
4     newelem->elem = elem;
5     newelem->next = m_top;
6     m_top = newelem;
7 }
8
9 template <typename T>
10 void Stack<T>::pop() {
11     if (NULL == m_top) return;
12     StackElem *topelem = m_top;
13     m_top = m_top->next;
14     delete topelem;
15 }

```

**Вопрос к студентам:** чего здесь явно не хватает?

Давайте посмотрим как применять разработанный шаблонный класс:

```

1 Stack <int> intstack;
2 Stack <double> dblstack;
3 intstack.push(2);
4 dblstack.push(2.0);
5 cout << intstack.top() << " " << dblstack.top();
6 intstack.push(3);
7 dblstack.push(3.0);
8 cout << intstack.top() << " " << dblstack.top();
9 intstack.pop();
10 dblstack.pop();
11 cout << intstack.top() << " " << dblstack.top();

```

Из примера ясно, что с использованием шаблонов, поддержка двух разнородных контейнеров (трёх, десяти) это крайне легко. Скорость компиляции, правда, страдает, вместе с объёмом кода, но по современным реалиям это невысокая цена.

**Вопрос к студентам:** эффективен ли этот стек для целых чисел?

#### 4.2.1 Специализация

Итак, хотелось бы получить отдельный стек из прошлого примера для целых чисел, чтобы сделать его гораздо эффективней.

В случае с функциями был использован механизм перегрузки функций чтобы получить специальное поведение для конкретного типа аргументов. Но нельзя “перегрузить” класс. Зато вы можете специализировать шаблон класса для конкретных шаблонных параметров. Синтаксис, опять же, прост:

```

1 template <>
2 class Stack<int> { .... };

```

Такая запись создаёт отдельное поведение для стеков, параметризованных целыми числами. Реализация и поведение такого шаблона могут не иметь вообще ничего общего с исходным. Это создаёт почти такую же радость при чтении кода на C++ как перегрузка операторов и неявные приведения.

Важным является тот факт, что специализация всегда должна физически в коде следовать общему шаблону (C++11, 14.7.3/3) и поэтому

нельзя сначала написать специализированную версию, а потом общую – общая версия должна быть хотя бы **объявлена** раньше.

Функции тоже могут быть специализированы (как и перегружены), что создаёт известную ортогональность. По правилам C++ вторая перегрузка выигрывает у первой, а потом специализация выигрывает у оригинала:

```

1 template <typename T> void foo(T);
2 template <typename T> void foo(T*);
3
4 // specialization of both foo(T) and foo(T*)
5 template <> void foo<int>(int*);
6
7 foo(new int); // calls foo<int>(int*);
```

Но тут важно, что компилятор засчитывает специализацию для типов, уже встретившихся ему внутри единицы трансляции. Поэтому (это известно как “контрпример Димова-Абрамса”):

```

1 template <typename T> void foo(T);
2 // specialization of foo(T)
3 template <> void foo<int*>(int*);
4 template <typename T> void foo(T*);
5
6 foo(new int); // calls foo(T*) !!!
```

В этом примере вторая перегрузка выигрывает у первой, после чего специализация вообще не рассматривается.

Стоп, скажете вы, как же так может быть, что явно более специальная версия функции с подходящими по типу аргументами оказывается выброшенной? Очень просто: специализации не участвуют в перегрузке. Комитет по стандартизации официально решил, что смена лидера перегрузки после специализации шаблона будет вводить программистов в заблуждение (недоуменное молчание, затем – общий смех). На самом деле ничего смешного тут нет, это решение крайне логично, просто, чтобы понять эту логику, нужно понимать как инстанцируются шаблоны, в частности знать о SFINAE см. (4.8.3).

Конечно вы не хотите таких эффектов в своей программе, поэтому там где можно перегрузить функцию, можно и нужно её перегружать, а не специализировать.

**Вопрос к студентам:** Бывают ли случаи когда необходимо все-таки

специализировать функцию?

Кроме того, начиная с C++11, конкретные специализации функций можно запрещать, так же как и перегрузки с конкретными параметрами (см. 2.6.2).

```

1 // for any ptrs
2 template <typename T> void foo(T*);
3
4 // except void* and char*
5 template <> void foo<void>(void *) = delete;
6 template <> void foo<char>(char *) = delete;
```

Вряд ли этой техникой вы будете пользоваться слишком часто, но кто знает.

**Вопрос к студентам:** являются ли целые числа единственным по чему хотелось бы специализировать стек? Как насчёт указателей?

#### 4.2.2 Частичная специализация

Да, действительно, приведённый выше стек хотелось бы специализировать для всех типов вида  $T^*$ . Для этого в языке есть техника, которая называется “частичной специализацией”.

Суть частичной специализации в том, что вы пишете отдельный код для случая когда только часть параметров исходного шаблона зафиксирована в некоторых значениях (или ограниченая некоторыми условиями). При исходном шаблоне

```
1 template <typename T, typename U> class MyClass { .... };
```

Возможно написать, например, такие специализации

```

1 // partial specialization: T == U
2 template <typename T> class MyClass<T,T> { .... };
3
4 // partial specialization: U == int
5 class MyClass<T,int> { .... };
6
7 // partial specialization: T == T1*, U == T2*
8 template <typename T1, typename T2>
9 class MyClass<T1*,T2*> { .... };
```

И далее при подстановке конкретных типов будет инстанцирована та или иная специализация.

```

1 MyClass<int,float> mif; /* uses MyClass<T1,T2> */
2 MyClass<float,float> mff; /* uses MyClass<T,T> */
3 MyClass<float,int> mfi; /* uses MyClass<T,int> */
4 MyClass<int*,float*> mp; /* uses MyClass<T1*,T2*> */

```

Учтите, что если под некое объявление подходят сразу два варианта частичной специализации, то это ошибка

```

1 MyClass<int,int> m; // ERROR: matches MyClass<T,T>
2 // and MyClass<T,int>
3 MyClass<int*,int*> m; // ERROR: matches MyClass<T,T>
4 // and MyClass<T1*,T2*>

```

Можно исправить ошибку сделав уникально подходящий шаблон

```

1 template <typename T>
2 class MyClass<T*,T*> { .... };

```

Очень важно понимать, что частичная специализация не доступна для функций и является прерогативой шаблонов классов.

Сэттер в известной публикации <http://www.gotw.ca/publications/mill17.htm> предлагает способ разрешить нечто вроде частичной специализации для функций:

```

1 template <typename T> struct FImpl;
2
3 template <typename T> void f(T t) {
4     FImpl<T>::f(t);
5 }
6
7 template <typename T> struct FImpl {
8     static void f( T t );
9 };

```

Частичная специализация в этом случае очевидно доступна для класса `FImpl` и напрямую влияет на зависимый тип статической функции-члена.

Но в большинстве случаев аналог частичной специализации для функций достигается перегрузкой.

**Дискуссия в аудитории:** вы – член комитета стандартизации и решаете сделать ли частичную специализацию доступной для функций. Ваши аргументы за и против?

**Домашняя наработка:** допишите частично специализированный для указателей стек

### 4.2.3 Шаблонные параметры по умолчанию

Некоторые из шаблонных параметров могут иметь значения по умолчанию.

```
1 // Case 1: default parameters
2 template <typename Key, typename Value = bool>
3 class KeyValuePair;
4
5 // ...
6
7 KeyValuePair<int> kvp;
```

Это очень похоже на специализацию. Если специализировать тот же шаблон для `bool`, использование в коде будет таким же или похожим:

```
1 // Case 2: template partial specialization
2 template <typename Key, typename Value>
3 class KeyValuePair;
4
5 template <typename Key>
6 class KeyValuePair<Key, bool>;
7
8 // ...
9
10 KeyValuePair<int> kvp;
```

Но это очень разные вещи. Специализация создаёт новый шаблон, который всегда выигрывает у шаблона с параметрами по умолчанию как у менее специализированного.

### 4.2.4 Вывод типов против подстановки типов

Ранее (см. 3.5.5) уже обсуждались возможности автоматического вывода типов, в частности для организации универсальных ссылок:

```

1 int x;
2 auto &&y = x;

```

**Вопрос к студентам:** вспомнить механизм вывода типов. Какой тип будет выведен для `y`?

Подобные техники возможны и для шаблонов:

**Вопрос к студентам:** в коде ниже аргумент `push_back` это правая или универсальная ссылка?

```

1 template <typename T, class Allocator = allocator<T>>
2 class Stack {
3 // .... some stack impl ....
4 public:
5     void push_back(T&& x);
6 };
7
8 int x;
9 Stack <int> s;
10 s.push_back(x);

```

Но очень важно отличать механическую **подстановку** типа в шаблонах от гораздо более сложного **вывода** типов (который бывает в тех же шаблонах, например шаблонная функция вполне способна вывести тип аргумента без явного указания).

#### 4.2.5 Разрешение имён

Шаблоны имеют дело с именами – заменой имени для типа, выводом имени, разрешением имени. Но должно ли разрешение имён в шаблонах (в том числе классов) происходить до инстанцирования или после?

```

1 template <typename T> struct Foo {
2     int use() { return illegal_name; }
3 };

```

Здесь `illegal_name` выглядит нелегальным именем, но может быть оно будет как-то легализовано после того как будет подставлен конкретный тип `T`? Нужно ли выдавать ошибку сразу или подождать подстановки параметра?

Ответ на эти вопросы даёт двухфазный поиск имён (two-phase name lookup).

- Первая фаза: до инстанцирования. Шаблоны проходят общую синтаксическую проверку, а также разрешаются независимые имена
- Вторая фаза: во время инстанцирования. Происходит специальная синтаксическая проверка и разрешаются зависимые имена

Зависимое имя – это имя, которое семантически зависит от шаблонного параметра. Шаблонный параметр может быть его типом, он может участвовать в формировании типа и так далее.

Теперь очевидно, что в примере выше было использовано независимое имя и, таким образом, речь идёт об ошибке первой фазы трансляции.

Но если сделать имя зависимым, ситуация меняется

```
1 template <typename T> struct Foo {
2     int use () { return T::illegal_name; }
3 };
```

Здесь первая фаза пройдёт гладко и мягко, а вторая будет зависеть от того есть ли в T в действительности `illegal_name`.

Мнемоническое правило для запоминания: **разрешение зависимых имен откладывается до подстановки шаблонного параметра**.

Потренируемся:

```
1 // template for foo
2 template<typename T> void foo (T) {
3     cout << "T";
4 }
5
6 struct S { };
7
8 // t is dependent from T, s -- not
9 template<typename T> void call_foo (T t) {
10     S x;
11     foo (x);
12     foo (t);
13 }
14
15 // non-template overload
16 void foo (S) {
17     cout << "S";
18 }
```

```

19
20 int main () {
21     S x;
22     call_foo (x);
23 }
```

**Вопрос к студентам:** что на экране?

Полезный пример из [11] даёт обратную картину: здесь `Base::exit` ещё не существует, а имя функции `exit` независимо от шаблонного параметра, поэтому оно “схватит” первую попавшуюся, скорее всего глобальную функцию `exit`.

```

1 template <typename T>
2 class Base {
3     public:
4         void exit();
5 };
6
7 template <typename T>
8 class Derived : Base<T>
9 {
10    public:
11        void foo()
12    {
13        // calls external exit() or error!
14        exit();
15    }
16};
```

**Вопрос студентам:** что же делать?

Итак, я надеюсь, теперь все стало понятнее. Если нет, то вам не повезло.

В последнем примере говорилось про `Base::exit`, а не про `Base<T>::exit`, потому что для удобного оперирования именами в специализациях и шаблонах разрешено их сокращенное употребление:

```

1 template <class T> class A {
2     A* a1; // A refers to A<T>
3 };
4
5 template <class T> class A<T*> {
```

```

6   A* a2; // A refers to A<T*>.
7 }

```

К счастью, классы нельзя перегружать, иначе все вообще бы запуталось.

#### 4.2.6 Устранение неоднозначности

Зависимые имена порождают неприятные проблемы, в основном связанные с тем какую собственно сущность обозначает зависимое имя.

Канонический пример приведён ниже.

```

1 template <typename T>
2 int foo (const T& x)
3 {
4     T::subtype *y;
5     // .... other code ....
6 }

```

Что здесь написано? Думаете объявление указателя на вложенный тип? Вы не поверите, но дословно здесь написано следующее: “В классе `T` должен быть статический член с именем `T::subtype`. Необходимо умножить его на некую (очевидно глобальную) переменную `y` и проигнорировать результат”. Печально, да? Давайте исправим код:

```

1 template <typename T>
2 int foo (const T& x)
3 {
4     typename T::subtype *y;
5     // .... other code ....
6 }

```

Теперь всё хорошо. Компилятор, благодаря явному указанию, понимает, что `T::subtype` это тип. Именно для этого `typename` и был запланирован. По английски устранение такой неоднозначности называется “**disambiguation**” и именно так этот термин устоялся в англоязычной литературе (не встречал его в русскоязычной). То, что я перевожу `disambiguation` как “устранение неоднозначности” это некая вольность, скорее передающая смысл термина, чем его перевод.

Более сложными для устранения неоднозначности являются зависимые имена шаблонов.

```

1 template<typename T> struct S {
2     template<typename U> void foo(){}
3 };
4
5 template<typename T> void bar() {
6     S<T> s;
7     s.foo<T>();
8 }

```

Попробуйте не читая дальше догадаться какую проблему содержит этот код?

Готово? Теперь проверьте себя.

Для ответа на этот вопрос обратим внимание на `s.foo<T>`. Что значит открывающая треугольная скобка после `foo`? Звучит неожиданно, но без разрешения неоднозначности первая треугольная скобка означала бы оператор меньше.

Для разрешения неоднозначности в зависимом имени шаблона используется ключевое слово `template`

```

1 template<typename T> void bar() {
2     S<T> s;
3     s.template foo<T>();
4 }

```

Время для небольшой тренировки. Совместное использование двух рассмотренных ключевых слов иногда критично для уточнения грамматики. Пусть задан код следующего вида.

```

1 template <typename Type> struct Outer {
2     template <typename InType> struct Inner {
3         template <typename X> void nested() {
4             printf("victory\n");
5         }
6     };
7 };

```

Необходимо написать вызов метода `nested`, параметризованного типом `int` из класса, в свою очередь параметризованного двумя независимыми типами (один из них можно использовать как `Type`, а второй как `InType`).

```

1 template <typename T1, typename T2> struct Usage {

```

```

2 void caller( /* parameter type */ &obj ) {
3     /* obj.nested<int>(); */
4 }
5 };

```

Можно написать простую проверочную программу, где использовать этот вызов как-то так:

```

1 Usage<int, int> u;
2 Outer<int>::Inner<int> obj;
3 u.caller(obj);

```

Сам по себе этот пример представляет собой лишь изящную головоломку, но полезен для иллюстрации концепции. В реальных проектах эти проблемы могут всплыть в гораздо более сложном коде и нужно вдоволь натренироваться на простых примерах, чтобы уметь их угадывать. Правильный ответ выглядит несколько необычно:

```

1 void caller(typename Outer<T1>::template Inner<T2> &obj) {
2     obj.template nested<int>();
3 }

```

Но что произойдёт если забыть ключевое слово `template`? Примерно та же проблема, что и с `typename` – компилятор не будет в точности знать как ему трактовать треугольную скобку, открывающуюся после имени типа `Inner` или после имени функции `nested`. И в том и в другом случае, без явного указания ключевого слова `template`, он будет трактовать эту скобку как переопределённый оператор “меньше” (снова нечто невероятное). Это может привести вас к массе неявных но интересных ошибок.

#### 4.2.7 Целочисленные шаблонные параметры

Кроме типов, шаблоны могут быть параметризованы целыми числами. Простейший пример: массив с заданным во время компиляции размером:

```

1 template<unsigned int S>
2 class Array {
3     unsigned char bytes[S];
4 public:
5     /* work with bytes */
6 };

```

Теперь в своем коде вы можете написать:

```
1 Array<42> a;
```

Объявив таким образом массив из сорока двух байт как класс с поведением, но без выделения динамической памяти.

Разумеется никто не мешает вам его специализировать.

```
1 template<>
2 class Array<3> {
3     /* alternative definition for SIZE == 3 */
4 };
```

Важно, что константа должна быть известна на этапе компиляции. Вы не можете написать нечто вроде:

```
1 int x;
2 // nontrivial code around x ....
3 Array<x> a; // Error!
```

Это ограничивает целочисленные параметры целочисленными константами **времени компиляции**.

**Вопрос к студентам:** можете ли вы обойти предыдущее ограничение с помощью `const int`?

```
1 int x;
2 // nontrivial code around x ....
3 const int cx = x;
4 Array<cx> a; // Ok?
```

Тут вроде бы мы параметризовали константой... Всё хорошо?

#### 4.2.8 Параметризация шаблонов указателями и ссылками

Параметризовать шаблоны можно указателями и ссылками (и левыми и правыми), но, опять-таки только если объекты на которые они указывают/ссылаются и сама связь объекта-рефери со своей ссылкой являются известными на этапе компиляции. Обычно это ограничивает параметризацию указателями и ссылками на глобальные переменные (что не слишком полезно) и указателями на свободные функции (а вот это бывает полезно и полезно КРАЙНЕ)

Сначала о переменных. Допустим, есть необходимость сопоставить каждой глобальной переменной в программе её имя и запомнить значение.

```

1 template <int *foo> struct VarNamer {
2     static const char *name;
3 };
4
5 int baz = 7;
6
7 template<>
8 const char * VarNamer<&baz>::name = "baz";
9
10 int bar = 42;
11
12 template<>
13 const char * VarNamer<&bar>::name = "bar";

```

Теперь для каждого глобала можно распечатать его имя и текущее значение

```

1 template <int *foo> void
2 detect_global ()
3 {
4     cout << VarNamer<foo>::name
5         << " = " << *foo << endl;
6 }

```

Эта штука работает не мощнее, чем обычная стрингификация в пре-процессоре, но требует указания имён вручную, так что на самом деле не слишком полезна (зато демонстрирует идею).

Поговорим о полезных штуках.

Очень часто в объект класса следует передать некий указатель на функцию, которой этот объект далее пользуется как callback функцией.

```

1 class CBack {
2     void (*callback_)();
3 public:
4     CBack(void (*callback)()) : callback_(callback) {}
5     void use() {
6         callback_();
7     }

```

```
8 };
```

В коде это можно использовать как-то так:

```
1 void foo() { cout << "callback" << endl; }
2 // .... later
3 CBack c(&foo);
4 c.use();
```

Если посмотреть внимательно, очевидно, что на этапе компиляции уже известно, что этот объект будет создан с этим callback и на протяжении всей программы это вряд ли изменится. Логично вынести это решение на этап компиляции. Теперь класс может выглядеть как-то так:

```
1 template<void (*callback_)()>
2 struct CBack {
3     void use() {
4         callback_();
5     }
6 };
```

Это существенное упрощение и оптимизация. Использующий его код при этом почти не меняется.

```
1 void foo() { cout << "callback" << endl; }
2 // .... later
3 CBack<&foo> c;
4 c.use();
```

Про более интересный случай речь пойдёт после знакомства с идиомой CRTP см. (4.4.2).

Ещё одно применение указателей на функции – получение шаблона, который делает автоматическую мемоизацию.

Предположим, что `cache` это объект класса (можно сделать его статическим внутри `memoize`) который поддерживает функции `find`, `val` и `store` с очевидной семантикой.

```
1 template <int (*f)(int)> int memoize(int x) {
2     if (cache.find(x)) return cache.val(x);
3     return cache.store(f(x));
4 }
5
6 int fib(int n) {
```

```

7   if (n < 2) return n;
8   return memoize<fib>(n - 1) + memoize<fib>(n - 2);
9 }
```

Эти числа Фибоначчи на удивление эффективны (особенно если правильно подобрать структуру данных для cache).

#### 4.2.9 Шаблоны псевдонимов

До сих пор двумя лучшими (воображаемыми) друзьями программиста были `typedef` (2.1.5) и `typename` (4.2.6). Но теперь оказывается, что в новом стандарте у `typedef` есть младший братик `using`

```

1 typedef int MyInt;
2 using MyInt = int;
```

Эти две строчки совершенно эквивалентны. Зачем же нужно было вводить новое ключевое слово? Потому что `using` умеет больше:

```

1 template <typename T>
2 using MyType = AnotherType< T, MyAllocatorType >;
3
4 template <typename T>
5 using ptr = T*;
6 ...
7 MyType<int> a;
8 ptr<int> x;
```

Очень удобно совмещать новый `using` с определителями типов. Например такое переопределение, как приведенное ниже:

```
1 template <typename T> using decay_t = typename decay<T>::type;
```

Позволяет сделать in-place сгнивалку для типов. Синонимы для всех распространенных определителей уже включены в стандарт.

В вашей стандартной библиотеке даже базовые типы могут быть определены исходя из реалий архитектуры:

```

1 using size_t = decltype(sizeof(0));
2 using ptrdiff_t = decltype((int*)0 - (int*)0);
3 using nullptr_t = decltype(nullptr);
```

Кстати, это даёт замечательный пример совмещения `using` и `decltype`

**Обсуждение в аудитории:** почему не был расширен синтаксис уже имевшегося `typedef`, а было перегружено ключевое слово `using` которое вообще-то совсем о другом?

Тема и впрямь дискуссионная. Имеется частное мнение Саттера, что “there is no type to define” и `typedef` показался комитету семантически непригодным для выражения идеи семейства типов. Ну ок.

## 4.3 Шаблоны и ООП

*These two have the same origin, but they differ in name*

*Both are called Mystery*

*– Laozi*

Самые коварные места в языке – это места сочленения различных его подсистем. Большинство языков программирования не являются мультипарадигменными и лишены этих проблем, но C++ это большой сложившийся конгломерат самых разных техник. На стыках этих техник часто возникают ситуации конфликта и взаимодействия. Особенно это касается использования возможностей ООП с учетом особенностей шаблонных классов. Как шаблоны соотносятся с наследованием, полиморфизмом, инкапсуляцией? Это неочевидно. Особенno интересна техника CRTP, когда шаблонный параметр причудливо переплетается с иерархией классов. Обо всем этом пойдет речь в этой лекции.

### 4.3.1 Специализация методов класса

Следующий пример вводит два класса: `Data` это не шаблонный класс с шаблонным методом `read<T>` и `DataReader<T>` это шаблонный класс с шаблонным методом `read<R>`

```

1 struct Data {
2     template <typename T> T read() const;
3 };
4
5 template <typename T>
6 class DataReader {
7     const T& source_;
8 public:
9     DataReader(const T& s) : source_(s) {}
10    /* internally calls source_.read() */
11    template <typename R> R read();
12 };

```

Можно написать реализацию метода `DataReader<T>::read<R>` вне тела класса, это довольно простое упражнение:

```

1 template <typename T>
2     template <typename R>

```

```

3 R DataReader<T>::read()
4 {
5     R foo = source_.read<R>();
6 }
```

На самом деле это не лучший вариант, потому что оставляет неоднозначность в синтаксисе. Её лучше сразу разрешить:

```
1 R foo = source_.template read<R>();
```

Гораздо сложнее написать специализацию этого метода. Сложно даже поставить задачу. Например задача написать специализацию вида `DataReader<T>::read<string>` сразу поставлена некорректно. Специализировать метод можно только если полностью специализирован также его класс (иначе это было бы эквивалентно частичной специализации функций а она запрещена). Поэтому корректная задача это написать например `DataReader<Data>::read<string>`. Она уже может быть решена:

```

1 template <>
2 template <>
3 std::string DataReader<Data>::read() const
4 {
5     std::string foo = " ";
6     return foo;
7 }
```

Немного вычурная конструкция `template <> template <>` при этом обязательна.

### 4.3.2 Переходники типов

Иногда не хочется писать специализацию или частичную специализацию всего класса ради того, чтобы изменить поведение одного нешаблонного метода. Если метод можно перегрузить, все хорошо. Но что если метод требует специализации (например у него просто нет аргументов подходящего для перегрузки типа)?

Задача: пусть один и тот же метод `func` класса `<T1, T2> class A` должен по разному себя вести если `T1` это целое число и если нет.

```

1 A <int, double> a;
2 A <float, double> b;
3
```

```

4 a.func(); /* for int */
5 b.func(); /* forall */
6 a.bar(); /* forall */
7 b.bar(); /* forall */

```

**Вопрос к студентам:** как этого добиться?

Первый вариант ответа может быть примерно такой:

```

1 template <typename T1, typename T2> class A {
2 public:
3     void func() {
4         T1 dummy; internal_func (dummy);
5     }
6 private:
7     template <typename V>
8     void internal_func(V) { cout << "forall" << endl; }
9
10    void internal_func(int) { cout << "forint" << endl; }
11 };

```

Но тут есть проблема – заранее неизвестно, что именно делает конструктор объекта класса T1. Стандартный выход из ситуации – подход с использованием переходника:

```

1 template <typename T> struct Type2Type {
2     typedef T OriginalType;
3 };

```

Подход с переходником позволяет не закладываться на конструктор класса T1 и сделать максимально легкий псевдо параметр.

```

1 template <typename T1, typename T2> class A {
2 public:
3     void func() { internal_func (Type2Type<T1>()); }
4 private:
5     template <class V>
6     void internal_func (Type2Type<V>)
7         { cout << "forall\n"; }
8
9     void internal_func (Type2Type<int>)
10        { cout << "forint\n"; }
11 };

```

Особое внимание нужно обратить на то, что переходник в таком виде имеет минимально возможный размер. Это вопрос ещё возникнет при разборе СРТР (см. 4.4), где станет ясно как можно обойтись вообще без переходника, используя лишь параметризованное наследование.

### 4.3.3 Шаблоны членов и инкапсуляция

Как видно из примера выше, в принципе шаблонная функция тоже может быть членом класса. Это не вносит почти никаких изменений в рассмотренный материал – для неё верно всё то же, что и для других шаблонных функций. Но использование шаблонных членов вносит серьёзные корректизы в инкапсуляцию. Коротко говоря, использование **открытых** шаблонных членов отменяет инкапсуляцию (шаблонные члены в закрытой части нормальны, безопасны и даже иногда неизбежны). Но как открытый шаблонный метод может вредить инкапсуляции данных? Через специализацию. Рассмотрим класс:

```

1 class X {
2 public:
3     X() : private_(1) {}
4     template <typename T> void f(const T& t) { /* ... */ }
5     int Value() const { return private_; }
6 private:
7     int private_;
8 };

```

За счёт наличия в этом классе шаблонного члена, человек, использующий его, может написать код:

```

1 namespace {
2     struct Y {};
3 }
4
5 template<>
6 void X::f(const Y&) {
7     private_ = 2;
8 }

```

Этот код эксплуатирует оговорённую в стандарте возможность специализировать шаблон-член для любого типа. То, что шаблоны-члены неявно нарушают инкапсуляцию – последствие взаимодействия объектно ориентированной и шаблонной подсистем языка, которые проектированы

вались во многом ортогонально и имеют нюансы совместного использования.

**Вопрос к студентам:** а как насчёт шаблонных членов и полиморфизма? Видите ли вы какие-нибудь проблемы в приведенном ниже коде?

```

1 template <typename T>
2 class Dynamic {
3 public:
4     virtual ~Dynamic();
5     template <typename T2> virtual void copy (T2 const&);
6     // .... etc ....
7 };

```

#### 4.3.4 Шаблоны полей и наследование

Ранее было упомянуто (3.7.4), что композиция и закрытое наследование существенно взаимозаменяемы. В шаблонном коде это тоже соблюдается. Можно написать либо композицию:

```

1 template <typename T1, typename T2>
2 class MyClass {
3     T1 a;
4     T2 b;
5 };

```

Либо закрытое наследование:

```

1 template <typename T1, typename T2>
2 class MyClass : private T1, private T2 {
3 };

```

С одной стороны вариант с закрытым наследованием даже предпочтителен если базовые типы пустые. Благодаря довольно известной технике empty base class optimizations (EBCO), получаем выигрыш одного байта против двух.

Увы, с другой стороны этот вариант плох. Например когда один из типов это `int`, он не будет работать.

Таким образом, в случае с шаблонами и обобщенным кодом, композиция должна заменяться на закрытое наследование с осторожностью.

### 4.3.5 Параметризация виртуальности

С помощью шаблонного наследования можно параметризовать виртуальность или невиртуальность функции, поскольку она зависит от определения функции в базовом классе:

```

1 struct NotVirtual {};
2
3 struct Virtual {
4     virtual void foo() = 0;
5 };

```

Здесь класс `NotVirtual` очень поучителен. Важно не то, что в нем есть, а то, чего в нем нет. Теперь иерархия с этими классами

```

1 template <typename VBase> struct Base : private VBase {
2     void foo() /* may be override */;
3 };
4
5 template <typename V> class Derived : public Base<V> {
6     void foo() /* may be override */;
7 };

```

Использование очевидно и даже красиво:

```

1 Base<NotVirtual>* p1 = new Derived<NotVirtual>;
2 p1->foo(); // ----- calls Base::foo()
3
4 Base<Virtual>* p2 = new Derived<Virtual>;
5 p2->foo(); // ----- calls Derived::foo()

```

Разумеется в этом примере параметризована виртуальность одной только функции `foo`, что, в общем, так себе. Может быть имеет смысл назвать класс `Virtual_foo` и в общем случае сделать макрос, порождающий имя `Virtual_X` для любого `X`.

**Домашняя наработка:** подумать над этим макросом и его применимостью.

### 4.3.6 Шаблоны и дружба

Изложение далее в целом будет следовать

<http://web.mst.edu/~nmjxv3/articles/templates.html>

TODO: само изложение. Идея в том, что дружить вот прямо так с шаблоном это слишком разрешительно и люди вертятся как могут.

Характерный изгиб ужа на этой сковородке:

```

1 template<typename T> class A;
2
3 template<typename T> A<T> foo(A<T>& a);
4
5 template<typename T>
6 class A
7 {
8     T m_a;
9 public:
10    A(T a = 0): m_a(a) {}
11    friend A foo<T>(A& a);
12 };

```

Но чаще просто объявляют шаблонную функцию-друга внутри шаблона класса.

В долгой истории шаблонов C++ встречаются примеры интересных обходных решений, изучение которых полезно не с практической а с общеразвивающей точки зрения. Одно из них, которое сейчас уже не актуально, решало интересную и важную проблему. Итак, перенесёмся в 1994-й и представим что у нас ещё нет перегрузки шаблонных функций и пространств имён.

Зато есть некий шаблон Array, для которого необходимо переопределить оператор сравнения на равенство `==`. Первый вариант – определить его как член класса, но это действительно плохая идея, поскольку первый аргумент (указатель на `this`) будет преобразоваться иначе чем второй. Такой пример уже разбирался в (3.3.2). Поэтому, конечно, его хочется определить вне класса:

```

1 template<typename T>
2 class Array {
3     // ...
4 };
5
6 template<typename T>
7 bool operator == (Array<T> const& a, Array<T> const& b)
8 {
9     // ...

```

10 }

Однако, если в языке нет перегрузки шаблонных функций, это создаёт проблему: ни одна большая функция с названием `operator ==` не может быть объявлена в этом же пространстве имён. Тем не менее, вполне логично, что оператор сравнения может быть нужен в том же пространстве имён для других классов. Бартон и Накман предложили определить этот оператор в классе как функцию-друг.

Эта техника получила название “трюк Бартона-Накмана”.

```
1 template<typename T>
2 class Array {
3     //...
4     public:
5         friend bool operator == (Array const& a, Array const& b) {
6             return ArraysAreEqual(a, b);
7         }
8     };
```

Пусть эта версия класса `Array` инстанцирована для типа `float`. Тогда оператор сравнения объявлен как результат этого инстанцирования, но сам по себе не является шаблоном функции и, таким образом, может быть перегружен как обычная функция.

В связи с тем, что `operator == (Array<T> const&, Array<T> const &)` определён внутри определения класса, он неявно считается встраиваемой функцией. Поэтому там и заложена делегация к не-инлайновой `ArraysAreEqual`, которая уже вряд ли будет конфликтовать с другими именами.

## 4.4 CRTP

Шаблонный класс после подстановки это уже настоящий класс и значит от него можно наследоваться. На этой технике основана интересная и часто используемая идиома CRTP.

### 4.4.1 Основная идея

**CRTP – curiously recurring template parameter** означает параметризацию шаблона, являющегося базовым классом в строчке объявления производного класса, шаблонным параметром, являющимся самим производным классом.

Это позволяет реализовать на чистых шаблонах механизм, похожий на обобщение виртуальных функций:

```

1 template <typename Child>
2 struct Base {
3     void interface() {
4         static_cast<Child*>(this)->implementation();
5     }
6 };
7
8 struct Derived : Base<Derived> {
9     void implementation() {
10        cerr << "Derived implementation" << endl;
11    }
12 };
13
14 template <typename T> void
15 call_interface (Base<T> *b) {
16     b->interface ();
17 }
```

Теперь вызов вроде бы как по указателю на базовый класс пройдёт гораздо более гладко и мягко

```

1 Derived d;
2 call_interface (&d);
```

Разумеется, список того, что может вызвать `Base` через свою функцию `interface` определяется программистом (в отличии от механизма

виртуальных функций, где вызов всегда пребрасывался до единственной функции производного класса с тем же именем).

Если вернуться к предыдущей задаче имитации специализированного метода, то её решение через CRTP вполне прозрачно:

```

1 template <typename S, typename T>
2 struct ABase {
3     void func() {static_cast<S*>(this)->forall(); }
4 };
5
6 template <typename S>
7 struct ABase <S, int> {
8     void func() { static_cast<S*>(this)->forint(); }
9 };
10
11 template <typename T1, typename T2>
12 struct A : public ABase <A<T1, T2>, T1> {
13     void forall () { cout << "forall\n"; }
14     void forint () { cout << "forint\n"; }
15 };

```

Главная проблема такого подхода – проблема с инкапсуляцией. Сейчас служебные методы `forint` и `forall` были размещены в открытой части класса, что, может быть, не лучшая идея.

Ключевой шаг к их закрытию выглядит так:

```

1 template <typename S, typename T> class ABase {
2     struct ACC : S {
3         static void access_forall (S* derived) {
4             void (S::*fn)() = &ACC::forall;
5             (derived->*fn)();
6         }
7     };
8     public:
9         void func() { ACC::access_forall(static_cast<S*>(this)); }
10 };

```

TODO: расписать эту идею подробнее (или оставить на самостоятельное исследование?)

Техника CRTP может быть использована не только для таких причудливых трюков, но и как вполне практичная замена для виртуальных

### функций

Например если задан произвольный класс Derived с уже определённым оператором меньше, можно в терминах этого оператора доопределить все остальные через шаблон Comparisons.

```

1 template <typename Derived> struct Comparisons {};
2
3 template <typename Derived>
4 bool operator==(const Comparisons<Derived>& o1,
5                 const Comparisons<Derived>& o2) {
6     const Derived& d1 = static_cast<const Derived&>(o1);
7     const Derived& d2 = static_cast<const Derived&>(o2);
8     return !(d1 < d2) && !(d2 < d1);
9 }
10
11 template <typename Derived>
12 bool operator!=(const Comparisons<Derived>& o1,
13                   const Comparisons<Derived>& o2) {
14     return !(o1 == o2);
15 }
```

Используя его через CRTP, можно бесплатно получить сравнения для пользовательского класса

```
1 class Person : public Comparisons<Person>
```

Возникает вопрос – зачем нужно CRTP, являющееся по сути имитацией динамического полиморфизма если у нас уже есть статический полиморфизм и те же сравнения могут быть написаны для любого T безо всякого CRTP? Идея в том, что CRTP позволяет ограничить область действия статического полиморфизма множеством наследников данного класса и в то же время избежать расходов на виртуальные функции.

Ещё одно интересное применение связано с виртуальным копированием. Идея виртуального конструктора уже рассматривалась в (3.6.7), но проблемой с таким подходом является то, что часто много таких конструкторов, крайне тривиальных, приходится писать вручную.

```

1 struct Vehicle {
2     virtual ~Vehicle() {}
3     virtual Vehicle *clone() const = 0;
4 };
5
```

```

6 struct Car : public Vehicle {
7     virtual Car *clone() const { return new Car(*this); }
8 };
9
10 struct Plane : public Vehicle {
11     virtual Plane *clone() const { return new Plane(*this); }
12 };

```

TODO: расписать

```

1 template <typename Base, typename Derived> struct MixClonable {
2     virtual Base *clone() const {
3         return new Derived(static_cast<Derived const &>(*this));
4     }
5 };
6
7 struct Car : public Vehicle, public MixClonable<Vehicle, Car>
8 {};
9 struct Plane : public Vehicle, public MixClonable<Vehicle,
10 Plane> {};

```

TODO: делегация конструкторов и MixClonableInh

#### 4.4.2 CRTP и связывание

Пусть есть шаблонная функция

```

1 template <typename X>
2 void foo (X *px);

```

И хочется написать шаблон класса с параметром T, параметризованный также `foo<T>`, берущей указатель на этот класс в качестве аргумента (что позволяет в частности передавать в неё `this`).

Базовое решение плохо тем, что требует рантайм присвоения и хранения указателя:

```

1 template <typename T>
2 struct X {
3     void (*pfoo_)(T*);
4     X(void (*pfoo)(T*)):pfoo_(pfoo){}
5     void call() { pfoo_(<this>); }
6 };

```

Это неэффективно и бесполезно, ведь нужной функцией можно параметризовать уже на этапе компиляции. Красивое решение через CRTP:

```

1 template <typename U, void (*pfoo)(U *pt)>
2 struct Xbase {
3     void call(U *pt) { pfoo(pt); }
4 };
5
6 struct X : Xbase<X, foo<X>> {
7     void call() { Tbase::call(this); }
8 };

```

CRTP очень полезно, чтобы выносить такую параметризацию на компилятор.

#### 4.4.3 Шаблонные шаблонные параметры

Представьте, что вы используете CRTP для предоставления “интерфейса” к набору дочерних шаблонов, при этом как родитель, так и потомки параметризованы:

```

1 template <typename derived, typename value> class interface {
2     void do_something(value v) {
3         static_cast<derived*>(this)->do_something(v);
4     }
5 };
6
7 template <typename value> class derived :
8     public interface<derived<value>, value> {
9     void do_something(VALUE v) { ... }
10 };
11
12 typedef interface<derived<int>, int> derived_t;

```

Строчка, определяющая `derived_t` содержит неприятное дублирование типа `int`, который на самом деле один и тот же и для потомка и для предка. Мало того это даёт возможность ошибки при опечатке. Чтобы явно отобразить, что шаблон параметризуется зависимым параметризованным типом:

```

1 template <template <typename> class derived, typename value>
2 class interface {

```

```

3     void do_something(value v) {
4         static_cast<derived<value>*>(this)->do_something(v);
5     }
6 };
7
8 template <typename value> class derived :
9     public interface<derived, value> {
10     void do_something(value v) { ... }
11 };
12
13 typedef interface<derived, int> derived_t;

```

Обратите внимание на `template <typename> class derived` – именно такой синтаксис внутри списка аргументов показывает, что `derived` это шаблонный тип, специфицированный неким зависимым.

Произвольное количество аргументов шаблона может стать аргументами шаблонного шаблонного параметра

```

1 template <template <typename> class Policy,
2         typename Gadget>
3 class GadgetManager: public Policy<Gadget>;
4
5 template <template <typename, typename> class Policy,
6         typename Widget,
7         typename WidgetPattern>
8 class WidgetManager: public Policy<Widget, WidgetPattern>;

```

Нет никаких проблем, чтобы растить параметр в ширину. Но также нет и проблем с тем, чтобы растить его в глубину.

```

1 template <typename T> struct Vector {};
2 template <template <typename> class Storage, typename Element>
3     struct Stack {};
4 template <template <template <typename> typename, typename>
5     class Stack,
6         template <typename> class Storage,
7         typename Element> struct StackMachine {};
8 Vector <int> v;
9 Stack <Vector, int> s;
10 StackMachine <Stack, Vector, int> a; // ОOOK!

```

**Домашняя наработка:** разберитесь со специализацией по шаблон-

ному шаблонному параметру.

#### 4.4.4 CRTP с закрытой базой и ещё немного дружбы

При использовании CRTP с закрытой базой, бывает полезно объявить закрытую базу другом, чтобы дать ей доступ к своей закрытой части.

Пусть по условию есть много конкретных транспортов, каждый из которых содержит метод с известным именем и сигнатурой:

```

1 template<typename Service>
2 class tcp {
3 public:
4     void send(...) {
5
6     }
7 };

1 template<typename Service>
2 class udp {
3 public:
4     void send(...) {
5
6     }
7 };

```

Тогда сервис, предназначенный для параметризации конкретным транспортом, но при этом сам транспортом не являющийся, может с одной стороны наследовать транспорту закрыто (отношение part-of) а с другой стороны объявлять его другом.

```

1 /* Transport<service> is implementation detail */
2 template<template<typename> class Transport>
3 class service : private Transport<service> {
4     /* since we derive privately, make the transport layer a
5      friend, so that it can cast its this pointer down */
6     friend class Transport<service>;
7 public:
8     typedef Transport<service> transport_type;
9     /* common code */
10    void do_something(...) {
11        send(...);

```

```
12     }
13 };
```

И именно это даёт ему возможность вызывать его методы как родные, одновременно пряча их от вызова извне.

Типичное применение:

```
1 typedef service<tcp> service_tcp;
2 typedef service<udp> service_udp;
```

**Вопрос к студентам:** как переписать `do_something` если закрытая база не сделана другом?

## 4.5 Вариабельные шаблоны

Вариабельные шаблоны расширяют идею шаблонов C++, подобно тому, как функции с произвольным числом аргументов расширяют возможности обычных функций. Прежде, чем рассматривать в деталях это новшество, следует сделать обзор использования троеточий (ellipsis) в C и C++, чтобы поместить новую информацию в контекст.

### 4.5.1 Поговорим о «...»

Троеточкия (ellipsis) впервые появились в языке C для таких функций, как `printf` и `scanf`

```
1 int printf (const char* format, ... );
```

Были придуманы общеизвестные приемы работы с ними – список актуально переданных аргументов, получаемый через `va_list` одновременно итерировался с помощью `va_arg`, причем никакой информации о типах не сохранялось и они были полностью отданы на откуп программиста (обычно использовалась та или иная форма дедуцирования из форматного списка). Это приводило к крайне тяжелым последствиям. Скажем простая строчка:

```
1 printf(name);
```

позволяла хакеру при её вызове со строчкой вроде `name = %f%f%f%x (%s)` распечатать содержимое рантайм-стека и вывести на экран некие хранящиеся там приватные для программы сведения. А наличие таких легальных возможностей, как форматный модификатор `%n`, позволявший модификацию переменных:

```
1 int i;
2 printf("abcde%nfghj\n", &i);
3 assert (i == 5); /* printed so far */
```

Делала такие функции любимым оружием злоумышленников.

Тем не менее, плюсы перевешивали минусы, а на атаки находились защиты. Впоследствии переменное число аргументов было распространено на макросы:

```
1 #define eprintf(...) fprintf (stderr, __VA_ARGS__)
```

В C++ всё это было сохранено, и, более того, сверху было добавлено использование троеточия в обработке исключений (см. 2.8).

```

1 try{
2     // Try block.
3 }
4 catch(...){
5     // Catch-all block.
6 }
```

Систематически прослеживается обращение с троеточием как с заменой слов “все что угодно”.

#### 4.5.2 Пачки параметров

В стандарте C++11, были введены две новых именованных конструкции:

- Пачкой параметров шаблона (template parameter pack) называется конструкция с лидирующим троеточием:

```
1 template<typename ... Args>
```

При этом такой шаблон может вначале иметь и не-вариабельную часть:

```
1 template<typename Arg, typename ... Args>
```

Можно рассматривать `... Args` как тип, означающий “типы всех остальных аргументов”, а первый тип как “тип первого аргумента в списке”.

- Конструкция с троеточием после неё называется пачкой параметров функции:

```

1 template<typename ... Args>
2 void f(Args ... args);
```

В вызове такой функции есть определенная свобода: можно передать любое количество аргументов разных типов или ни одного.

```

1 f();           // OK: args contains no arguments
2 f(1);         // OK: args contains one argument: int
3 f(2, 1.0);   // OK: args contains two arguments: int and
                 double
```

Для пачки параметров также сделали оператор `sizeof...` (`Args`) чтобы выяснить её размер на этапе компиляции. Здесь троеточие не имеет отношения к параметрам функции или шаблона, а имеет третий смысл, являясь частью имени оператора. Увы, пользовательские операторы такого вида делать нельзя. Важно запомнить: оператор `sizeof...` возвращает размер пачки параметров в штуках, а не в байтах.

**Вопрос к студентам:** как вы думаете, возможны ли в языке C++ две разных пачки типов в одном шаблоне

```
1 template<typename ... T, typename ... U>
2 void f(T ... argst, U ... argsu);
```

Главное, что можно сделать с пачкой параметров функции это раскрыть её

```
1 template<typename ... Types> void f(Types ... rest);
2 template<typename ... Types> void g(Types ... rest) {
3     f(rest ...); /* expander */
4 }
```

Здесь при вызове `g (1, 1.0)` будет инстанцирована функция `g (int x, double y)` и в ней сгенерирован вызов `f (x, y)`. Тут нужно заметить, что пачка параметров функции тоже в некотором смысле раскрывает пачку параметров шаблона (см. положение троеточия). Получается иерархия: пачка типов может быть раскрыта в пачку параметров, а пачка параметров раскрыта в конкретном месте использования. Но обратите внимание: конечный раскрыватель уже не именован. Такой же неименованный конечный раскрыватель может быть написан для пачки типов.

Итак, новый стандарт вводит четыре (!) новых смысла для троеточия: пачка параметров шаблона, пачка параметров функции, часть оператора `sizeof...` и раскрытие пачки параметров в коде. Остаток лекции будет посвящен тому, чтобы не путаться во всех четырех.

### 4.5.3 Паттерны раскрытия пачек параметров

В нормальном случае пачка параметров шаблона используется как тип пачки параметров функции и раскрывается в точке использования (например в точке рекурсивного вызова):

```
1 template<typename T>
2 T adder(T v) {
```

```

3     return v;
4 }
5
6 template<typename T, typename... Args>
7 T adder(T first, Args... args) {
8     return first + adder(args...);
9 }
```

Видно, что язык позволяет таким образом пробросить пачку параметров, это критично важный сценарий их применимости.

В качестве паттерна при раскрытии пачки параметров может быть не обязательно её имя, но и её имя с какими-нибудь добавками, как в случае с `const_cast` ниже:

```

1 template<typename ... Args> void
2 f(Args ... args)
3 {
4     g(&args...); // expands to g(&E1, &E2, etc)
5     g(const_cast<const Args*>(&args)...);
6     // g(const_cast<const E1*>(&X1), const_cast<const E2*>(&X2),
7         etc
7 }
```

Здесь выражается идея “привести все”. Аналогичный пример можно привести с `std::forward`

```

1 template<typename Arg, typename ... Args> void
2 foo(Arg &&arg, Args &&... args)
3 {
4     bar(std::forward<Arg>(arg));
5     foo(std::forward<Args>(args)...);
6 }
```

Можно сказать, что троеточие справа от чего-либо используется для расширения пачки аргументов по шаблону того, за чем оно стоит:

```

1 template<typename ... T>
2 void f(T ... args)
3 {
4     g( args... );      //pattern = args
5     h( x(args)... ); //pattern = x(args)
6     m( y(args... ) ); //pattern = args
7     n( z<T>(args)... ); //pattern = z<T>(args)
```

```
8 }
```

Именно поэтому оно может быть использовано даже в контексте класса для создания списка наследования.

**Вопрос к студентам:** как вы думаете, для приведенного ниже класса наследование от второго и далее предков из вариабельного списка будет открытым или закрытым?

```
1 template<typename ... Mixins>
2 class mixture : public Mixins ...
```

**Вопрос к студентам:** Будет ли ошибка компиляции, если вызвать такой сложный паттерн с пустой пачкой параметров?

И да, более того, можно привести ещё более хитрый пример из стандарта C++14, раздел 14.5.3.6. Что будет, если попытаться раскрыть нижеследующий код с пустой пачкой параметров?

```
1 template<typename... T> struct X : T... { };
2 template<typename... T> void f(T... values) {
3     X<T...> x(values...);
4 }
```

На удивление: и тут все будет хорошо. У структуры `X` магическим образом не окажется наследников, переменная `x` будет такой, как если бы была объявлена с value-инициализацией `X x();`. Несмотря на то, что механический пропуск в этом месте пачки параметров вызвал бы проблемы с грамматикой (подвисшее в воздухе двоеточие, опустевшие треугольные скобки), стандарт гарантирует, что пустая пачка параметров во всех случаях разворачивается в пустой список так, чтобы не изменить синтаксис охватывающего выражения.

Следует обратить особое внимание на:

```
1 X<T...> x(values...);
```

Здесь двойное раскрытие: и пачки типов и пачки аргументов. Это двойное раскрытие не будет декартовым произведением, оно будет внутренним произведением. Совместное раскрытие по таким правилам возможно производить даже в одно троеточие:

```
1 f(const_cast<const T*>(&values)...);
```

Точно так же раскрытие можно комбинировать с пробросом параметров:

```
1 f(h(args ...) + args ...);
```

#### 4.5.4 Emplace и доброе волшебство

Давно, ещё со времен стандартизации библиотеки, перед разработчиками обобщённого кода стояла проблема создания контейнера из тяжело копируемых объектов.

```
1 class Heavy {
2     int n;
3     int *t;
4 public:
5     explicit Heavy (int sz) : n(sz), t(new int[n]) {
6         cout << "Heavy created" << endl;
7     }
8
9     Heavy(const Heavy &rhs) : n(rhs.n), t(new int[n]) {
10        cout << "Heavy copied" << endl;
11        memcpy (t, rhs.t, n*sizeof(int));
12    }
13};
```

В качестве контейнера можно рассмотреть стек.

```
1 template <typename T>
2 class Stack {
3     struct StackElem {
4         T elem;
5         StackElem *next;
6         StackElem (T e, StackElem *nxt) : elem (e), next (nxt) {}
7     };
8
9     struct StackElem *top_;
10 public:
11     void push_back (const T& elem) {
12         StackElem *newelem = new StackElem (elem, top_);
13         top_ = newelem;
14     }
15};
```

Как создать стек из десяти таких тяжелых неинициализированных объектов, каждый размера 100?

```

1 Stack<Heavy> s;
2 for (int i = 0; i != 10; ++i)
3     s.push_back(Heavy(100));

```

Легко видеть одно создания и два копирования. На экране:

```

Heavy created
Heavy copied
Heavy copied
Heavy destroyed
Heavy destroyed

```

Проблема может быть решена с помощью идиомы **emplace** (создания на месте). Для этого вариабельные шаблоны работают вместе с правыми ссылками. Сначала emplace конструктор вводится в структуру элемента списка

```

1 struct StackElem {
2     ...
3     template< class... Args >
4         StackElem (StackElem *nxt, Args&&... args) :
5             elem(std::forward<Args>(args)...), next (nxt) {}
6 };

```

Далее в сам список вводится метод emplace:

```

1 template <typename T>
2 template< class... Args >
3 void Stack<T>::emplace_back( Args&&... args )
4 {
5     StackElem *newelem =
6         new StackElem (top_, std::forward<Args>(args)...);
7     top_ = newelem;
8 }

```

И наконец модифицируется место вызова

```

1 Stack<Heavy> s;
2 for (int i = 0; i != 10; ++i)
3     s.emplace_back(100);

```

Теперь вывод гораздо сильнее радует глаз:

Heavy created

**Вопрос к студентам:** зачем был использован `std::forward` в этом контексте?

#### 4.5.5 Раскрытие с помощью списка инициализации

Возможна ли некая обработка (например вывод на экран) всех параметров пачки без явной рекурсии? Это интересный вопрос, которому посвящен этот раздел.

Пусть есть некая функция:

```
1 template <typename T> decltype (auto) foo (T x);
```

Необходимо внутри другой функции

```
1 template <typename... T> void lvisualize(T... args)
```

Вызвать `foo` для каждого параметра из `args`.

Первая попытка могла бы выглядеть как-то так:

```
1 template <typename... T>
2 void lvisualize01(T... args)
3 {
4     int x[] {((foo(args), 0)... };
5 }
```

Здесь `x` это массив из нулей. Имя `x` кажется здесь лишним и можно использовать возможность создания с помощью списков инициализации безымянных переменных. Сама эта возможность была сделана для того, чтобы разрешить `ranged for`:

```
1 for(int x : {-1, -2, -3})
2     cout << x << endl;
```

Здесь `{-1, -2, -3}` это список инициализации, который создает временный объект, живущий до конца полного выражения. Используя это, можно сделать симпатичней:

```
1 template <typename... T>
2 void lvisualize02(T... args)
3 {
4     using expand_type = int [];
```

```

5     expand_type {((foo(args), 0)... };
6 }
```

Теперь массив из нулей живет только до конца строчки. Увы, нужно учесть ещё два фактора: во-первых вызов `lvisualize02` с пустой пачкой параметров приведет к появлению массива нулевой длины. Это запрещено стандартом и рассмотренная выше магия здесь не сработает, потому что никакая синтаксическая конструкция не сломана. Массив нулевой длины объявлен правильно, просто такие массивы не допустимы. Во вторых (и это более забавно) в том типе, который возвращает `foo` может быть переопределенный оператор запятая. Следующий подход к решению использует void-expression чтобы блокировать возможную запятую и расширяет инициализатор первым нулем.

```

1 template <typename... T>
2 void lvisualize03(T... args)
3 {
4     using expand_variadic_pack = int[];
5     expand_variadic_pack {0, ((foo(args)), void(), 0)... };
6
7     /* leading 0 for void parameter packs
8      uses braced-init-list initialization rules
9      void() is to prevent malicious "operator," overloads,
10     which cannot exist for void types */
11 }
```

На этом месте у всех должно сложится чувство, что всё пошло по какому-то неправильному пути. И это, конечно, так. Неправильным было самое первое решение взять конкретный тип `int` и согласие на (пусть и временное) существование потенциально огромного массива из нулей. Что действительно хочется, так это разработать такой тип `expand_type`, чтобы весь пробросчик свелся к чему-то очень простому:

```

1 template <typename... T>
2 void lvisualize(T... args)
3 {
4     expand_type {((foo(args), void(), 0)... };
5 }
```

**Вопрос к студентам:** как написать такой тип?

Студент может предложить следующий ответ:

```
1 struct expand_type
```

```

2  {
3      template <typename... T> expand_type(T&&...) {}
4  };

```

Казалось бы это все. Тип занимает один байт и содержит единственный конструктор, берущий аргументы по набору универсальных ссылок.

Универсальные ссылки, кстати, не так уж и нужны, там всё равно `int`. Но главная проблема существенно сложнее: в таком виде код может распечатать список аргументов **в любом порядке**.

**Вопрос к студентам:** и что же делать?

#### 4.5.6 Наконец-то безопасный printf

Используя эти возможности, можно соорудить безопасный относительно типов `printf` не подверженный, рассматривавшимся выше атакам. Основной шаг рекурсии контролирует наличие спецификатора для каждого из поданных аргументов

```

1 template<typename T, typename... Args> int
2 pp_printf(const char* s, const T& value, const Args&... args)
3 {
4     while (*s)
5     {
6         if (*s == '%' && *++s != '%')
7         {
8             /* ignore the character that follows the '%'! */
9             cout << value;
10            return printf (++s, args...) + 1;
11        }
12        cout << *s++;
13    }
14    throw runtime_error ("extra arguments provided");
15 }

```

Завершение рекурсии содержит контроль наличия всех необходимых аргументов.

```

1 int
2 pp_printf(const char* s)
3 {
4     while (*s)

```

```

5   {
6     if (*s == '%' && ++s != '%')
7       throw runtime_error ("missing arguments");
8     cout << *s++;
9   }
10  return 0;
11 }
```

Следует обратить особое внимание на то, что обе потенциально ошибочные ситуации явно возбуждают исключения. Также замена простого троеточия на более интересную конструкцию `const Args&... args` позволяет иметь не просто любое количество аргументов, но любое количество типизированных аргументов (что совсем другой разговор).

**Вопрос к студентам:** что нужно сделать, чтобы заработали такие модификаторы как `%n` и которые сейчас блокируются константностью ссылок?

**Домашняя наработка:** попробуйте, универсализовав ссылки, добиться работы `%n` и `/*s` в безопасном `printf`

#### 4.5.7 Наконец-то совершенно прозрачная оболочка

Собирая вместе информацию о пробросе выводимого возвращаемого типа (см. 2.11.4) и пробросе аргументов (см. 3.5.6), теперь можно написать совершенно прозрачную обертку над функцией:

```

1 template<typename Fun, typename... Args>
2 decltype(auto) transparent(Fun fun, Args&&... args) {
3   return fun (std::forward<Args> (args)...);
4 }
```

Эта обертка позволяет вызвать любую функцию передав ей любое количество аргументов, в точности сохранив их типы, и вернуть тот тип, который возвращает обернутая функция.

## 4.6 Кортежи

Стандартные кортежи являются отдельным и крайне полезным нововведением, непосредственно вытекающим из вариабельных шаблонов и прямо с ними взаимосвязанным.

### 4.6.1 Слияние ежа и ужа

Вариабельные шаблоны в некотором смысле “объединяют” аргументы разных типов, вводя в C++ возможность сделать кортежи. В отличии от структур имеющих именованные поля разных типов и массивов, имеющих неименованные поля одного типа, кортежи это такой вид синтаксического клея, который позволяет склеивать поля разных типов. Можно сказать, что это структура которая ведет себя как массив. Механическое склеивание разнотипных полей можно проиллюстрировать через compile-time отображение с переопределенным оператором, скажем `operator^` для доступа к полю по тегу поля:

```
1 auto person = ctmap (age, last_name);
2 person^last_name = "Smith";
3 person^age = 50;
```

Первый вопрос – что это могут быть за аргументы: `age`, `last_name` и так далее? Это и есть теги всевозможных типов, между которыми можно строить отображения:

```
1 template<typename T>
2 struct tag{using value_type=T;};
3
4 static struct: tag<int>{} age;
5 static struct: tag<std::string>{} last_name;
6 /* ... */
```

Чтобы реализовать кортежи из этих тегов, потребуется несложная структура с описанием поля и переопределенным оператором доступа:

```
1 template <typename T>
2 struct Field
3 {
4     typename T::value_type storage;
5
6     auto &operator^ (const T &)
```

```

7   {
8     return storage;
9   }
10 };

```

После такой подготовки, само отображение выглядит довольно просто

```

1 template<typename... Fields>
2 struct ctmap_t:
3   public Field<typename decay<Fields>::type>... {};

```

Здесь `operator^` наследуется свой от каждого базового типа. Для удобства можно сделать функцию-хелпер, позволяющую автоматическое создание отображения по заданным на вход аргументам.

```

1 template<class...Fields>
2 ctmap_t<Fields...> ctmap( Fields&&... ) { return {}; }

```

Делегирование значений полей через универсальные ссылки – обычна практика в таких случаях. Увы, даже на последнем GCC этот пример не откомпилируется без некоторых хаков, так как компилятор увидит ambiguity (которой на самом деле нет) в унаследованных операторах.

**Домашняя наработка:** как все-таки собрать compile-time map такого вида на GCC 5.1?

Ещё одна проблема: необходимость своего тега для каждого типа и невозможность двух полей с одним и тем же тегом.

#### 4.6.2 Стандартные кортежи и их применения

К склеиванию данных есть и стандартные подходы, более простые и надежные, чем рассмотренное выше compile-time отображение, но, увы, оставляющие поля кортежей неименованными. Стандартная библиотека определяет класс `std::tuple` для механического объединения разнотипных данных.

```
1 std::tuple<int, double, int> t(1, 2.0, 3);
```

Доступ к трем полям осуществляется рекурсивно через `base` и `head_`

```

1 assert (t.head_ == 1);
2 assert (t.base.head_ == 2.0);
3 assert (t.base.base.head_ == 3);

```

Кроме того, можно обращаться к полям по индексу, используя стандартную функцию `std::get`

```
1 assert (get<0>(t) == 1);
2 assert (get<1>(t) == 2.0);
3 assert (get<2>(t) == 3);
```

В стандарте C++14 добавлена возможность обращаться к полям по типу:

```
1 assert (get<double>(t) == 2.0);
```

Но попытка получить нечто по типу `int` в данном случае приведет к ошибке компиляции из-за неоднозначности.

Посредством функции `std::tie` удобно распаковать кортеж, например чтобы обработать возвращаемые из функции несколько аргументов:

```
1 std::tuple<int,int> foo();
2
3 /* ... */
4
5 int a;
6 int b;
7 std::tie(a,b)=foo();
```

Кроме `tie`, кортеж можно создать также функцией `make_tuple`. Существенная разница между этими функциями: если `tie` используется в основном для распаковки возвращаемых параметров, то `make_tuple` делает деградацию и развертывание ссылочных обёрток (`unrefwrapping`) своих аргументов.

Стандартная функция `std::tuple_cat` позволяет конкатенировать произвольное количество кортежей в один кортеж

```
1 std::tuple<int, std::string, float> t1(10, "Test", 3.14);
2 int n = 7;
3 auto t2 = std::tuple_cat(t1, std::make_pair("Foo", "bar"), t1,
4                           std::tie(n));
```

Следует обратить внимание на то, что `std::make_pair` – единственный способ создать нечто похожее на кортеж в C++98 – вполне принимается как способ создать кортеж в C++14.

При работе с парами, начиная с C++14 возможно конструирование по частям, и есть надежды, что скоро его распространят и на кортежи:

```

1 pair<vector<int>, vector<int>> p (
2     piecewise_construct,
3     forward_as_tuple(),
4     forward_as_tuple(1, 2, 3));

```

Создаёт пару из двух векторов.

В C++17 собираются ввести удобную функцию `make_from_tuple`, которая создаёт экземпляр указанного класса с пробросом его конструктору параметров, перечисленных в кортеже.

```

1 tuple <int, int> t (3, 42);
2 auto v = make_from_tuple < vector<int> > (t);

```

Полученный вектор состоит из трёх элементов каждый со значением 42, увы, так работает конструктор вектора с двумя аргументами.

### 4.6.3 Отложенное исполнение на кортежах

Ещё одна интересная задача: создание отложенного исполнения (для C++11)

Для примера какой-нибудь простой функции:

```

1 double foo(int x, float y, double z)
2 {
3     return x + y + z;
4 }

```

Её отложенное исполнение происходит по такому сценарию: запомнить кортеж её аргументов и саму функцию в объект специального класса, для которого потом вызвать отложенное исполнение.

```

1 std::tuple<int, float, double> t = std::make_tuple(1, 1.2, 5);
2 save_it_for_later<int, float, double> saved = {t, foo};
3 /* ... */
4 auto x = saved.delayed_dispatch();

```

Сначала базовые строительные блоки: последовательность номеров и генератор последовательностей.

```

1 template<int ...>
2 struct seq {};
3

```

```

4 template<int N, int ...S>
5 struct gens: gens<N-1, N-1, S...> {};
6
7 template<int ...S>
8 struct gens<0, S...>{ typedef seq<S...> type; };

```

Обертка для сохранения элемента отложенного исполнения теперь использует генератор последовательностей для генерации последовательности номеров и последовательность номеров для вытаскивания аргументов из шаблона, в котором они сохранены.

```

1 template <typename ...Args>
2 struct save_it_for_later
3 {
4     std::tuple<Args...> params;
5     double (*func)(Args...);
6
7     double delayed_dispatch()
8     {
9         return callFunc(typename gens<sizeof...(Args)>::type());
10    }
11
12    template<int ...S>
13    double callFunc(seq<S...>)
14    {
15        return func(std::get<S>(params) ...);
16    }
17 };

```

В этом примере можно заменить сложный `gens` на появившийся в C++14 `index_sequence`.

#### 4.6.4 Кортежи для обработки вариабельных шаблонных параметров

Метод итерации по аргументам шаблонной пачки, рассмотренный в (4.5.5) не так уж и хорош. Использование кортежей предполагает возможность просто сложить всю пачку параметров в кортеж и для этого есть удобное стандартное средство.

```

1 template <class... Types> tuple<Types&&...>

```

```

2 forward_as_tuple (Types&&... args)
3 {
4     return tuple<Types&&...>(
5         std::forward<Types>(args)...);
6 }
```

Например следующая функция возвращает значение третьего аргумента (самое сложное в ней это вывод типов).

```

1 template <class ... Args>
2 auto third_arg (Args&& ... args)
3     -> ElementType <2, std::tuple<Args ...> >
4 {
5     return std::get<2>(
6         std::forward_as_tuple(
7             std::forward<Args>(args)...));
8 }
```

**Вопрос к студентам:** чем может быть ElementType в приведенном коде?

Важные правила:

- Не задавать явных шаблонных параметров для `make_tuple`
- Не сохранять на будущее результат `forward_as_tuple`

Ещё один красивый пример: как получить хвост кортежа? Для этого понадобится стандартный шаблон `std::tuple_size` и немножко магии.

```

1 template <typename Tuple>
2 decltype(auto) get_back (tuple &&t) {
3     return get <
4         tuple_size <
5             remove_reference_t <Tuple>
6             >::value - 1> (forward<Tuple>(t));
7 }
```

Тут стоит обратить внимание на уместность возвращаемого типа.

## 4.7 Lambda expressions

Greenspun's Tenth Rule of Programming:

*Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp*

– Philip Greenspun

Ещё одной, непросто воспринимаемой для программиста с опытом С концепцией, являются lambda expressions ( $\lambda$ -выражения). Мир языка С строго императивен, он не предполагает абстракции функций в первоначальные сущности. С другой стороны, даже небольшой опыт таких языков как Lisp или Haskell позволяет иначе взглянуть на вычислимость как таковую и для людей с хотя бы поверхностным знанием любого функционального языка, этот раздел может показаться даже излишне простым.

В любом случае, в современном C++  $\lambda$ -выражения это ещё один инструмент и его нужно знать и уметь хотя бы читать код, написанный в таком стиле. Настоящее же осознание применимости этой концепции приходит с опытом.

Hello world с первого взгляда немного шокирует.

```
1 int main () {
2     return [] () -> int { puts ("Hello!\n"); return 0; } ();
3 }
```

Здесь, согласно стандарту (C++14 5.1.2):

- [] – capture specifier, определяет lambda-выражение. Внутри его скобок также можно задать захватываемый контекст.
- () – argument list, здесь можно задать список параметров
- -> int – означает что возвращаемое значение имеет тип int.
- Внутри {} – тело lambda-выражения.

Сам термин  $\lambda$ -expression уходит корнями в математическую логику (работы Алонсо Черча по основаниям математики), но ниже употребляется не в математическом смысле, а в смысле рассмотренной только что

синтаксической конструкции языка C++. Вообще опыт автора показывает, что знание  $\lambda$ -calculus для освоения этого материала скорее вредно. В этом смысле термины lambda-функция, лямбда-функция или просто лямбда также будут ниже свободно взаимозаменяемы.

Тот же Hello world можно переписать несколько более многословно:

```

1 // creating lambda-function
2 auto func = [] () -> int { puts ("Hello!\n"); return 0; };
3
4 // calling it
5 return func ();

```

Каким будет выведен тип `func`, пока не так существенно. По своему действию получившийся объект похож на функтор из (3.3.4). Рассмотренный пример использования  $\lambda$ -выражения, может быть переписан несколько проще.

```

1 auto func = [] { puts ("Hello!\n"); return 0; };
2 return func();

```

По стандарту, если список аргументов не указан, то он пуст (C++14 5.1.2.4) и возвращаемый тип выражения выводится из `return` в его теле, если он указан и единственный (там же).

$\lambda$ -выражения в их простейшем виде хороши для объявления на месте несложных функторов. Стандарт (C++14 5.1.2.1) приводит простой пример:

```

1 void abssort(float *x, unsigned N) {
2     sort(x, x + N, [] (float a, float b) {
3         return abs(a) < abs(b);
4     });
5 }

```

**Вопрос к студентам:** можно сделать из `abssort` просто синоним для нужного `sort` с использованием  $\lambda$ -expressions?

### 4.7.1 Generic lambdas

Первое приближение к  $\lambda$ -функциям в стандарте C++11 позволяло списки инициализации только с конкретными типами аргументов:

```

1 auto ifunc = [](int input) { return input * input; };

```

```

1 auto dfunc = [](double input) { return input * input; };
2 /* ... */

```

Но простое программирование на шаблонах оказалось в некотором смысле более мощным, так как позволяло создать обобщенную (type-generic) функцию (или функтор) и сделать это совсем несложно:

```

1 template <typename T>
2 T func(T z) { return z * z; }

```

Для того, чтобы  $\lambda$ -выражения не уступали в выразительной мощности, в стандарте C++14 появились обобщенные  $\lambda$ -expressions.

```

1 auto func = [](auto input) { return input * input; };
2 /* ... */
3 std::cout << func(2) << std::endl;
4 std::cout << func(2.0) << std::endl;
5 std::cout << func(std::complex<double>(2.0, 1.0)) << std::endl;

```

Они особенно удобны для создания функторов при использовании вместе с алгоритмами стандартной библиотеки.

Конечно, есть и проблемы. Допустим есть необходимость используя perfect forwarding (см. 3.5.6) протащить из полиморфной лямбды параметр в какую-то функцию:

```

1 auto forwardingLambda = [](auto&& param)
2 { f(std::forward<T>(param)); };

```

Проблема в том, что совершенно неясно что подставить в качестве T.

**Вопрос к студентам:** что бы вы подставили?

Итак, лямбды могут все, что могут обычные функции (для них даже можно сделать аналог перегрузки, см. 4.7.4), а обобщённые лямбды могут всё, что могут шаблонные функции. Но факт в том, что на самом деле лямбды могут больше.

## 4.7.2 Замыкания

Можно ли на C++ написать  $\lambda$ -функцию, которая возвращала бы собственный аргумент?

Да и легко:

```

1 auto Identity = [](auto x) {
2     return x;
3 };

```

Можно ли написать более сложную функцию, которая возвращала бы объект класса-функтора, который потом возвращал бы аргумент при каждом вызове?

Да, тоже можно:

```

1 auto identityf = [](auto x) {
2     class Inner {
3         int x;
4         public: Inner(int i): x(i) {}
5         int operator() () { return x; }
6     };
7     return Inner(x);
8 };

```

Здесь видна одна интересная идея: идея **захвата контекста**.

Можно ли написать теперь ещё более сложную функцию, которая возвращала бы другую функцию с тем же свойством что у класса-функтора в примере выше?

Оказывается тоже можно. Для этого, сконструированная внутри лямбды, лямбда должна захватить контекст:

```

1 auto unit = [](auto x) {
2     return [=]{}() { return x; };
3 };

```

Теперь даже при её использовании за пределами создавшего её контекста, она будет “помнить” значение аргумента в момент создания.

Такая функция, возвращающая функцию, созданную внутри себя называется **функцией высшего порядка** и **unit** – простейший пример такой функции (для ФВП она выполняет роль “единицы”).

Ещё одна интересная концепция, относящаяся к ФВП это **каррирование**. Каррированием называется создание функции по образцу исходной но с частично подставленными аргументами. Например функция  $add(x, y) = x + y$  может быть каррирована до  $add_4(x) = x + 4$ . Замыкания вместе с вариабельными шаблонами позволяют легко делать каррирование функций:

```

1 template<typename Function, typename... Arguments>
2 auto curry(Function function, Arguments... args) {
3     return [=](auto... rest) {
4         return function(args..., rest...);
5     }
6 }
```

Использовать это как-то так:

```

1 auto add = [](auto x, auto y) { return x + y; }
2 auto add4 = curry(add, 4);
```

Теперь функция `add4` может быть использована как нормальная функция.

Эта идея захвата контекста оказалась невероятно продуктивна и существенная часть возможностей языка была введена ради её поддержки. Следует очень хорошо разобраться с захватом контекста прежде чем двигаться дальше.

### 4.7.3 Захват контекста в подробностях

Список для захвата контекста пишется в квадратных скобках. Правила формирования списка для захвата контекста регулируются стандартом (C++14 5.1.2.8). Существует два специальных символа – `=` и `&`. Переменные захватываемые по значению, входят в список без модификаторов, захватываемые по lvalue reference – с модификатором `&`

```
1 [foo, &bar]
```

Употребление `&` означает, что “может быть захвачена по ссылке любая переменная из контекста”, тогда захватываемая переменная не должна предваряться `&` и всё равно будет захвачена по ссылке (по ссылке значит по lvalue ссылке, говоря точно).

```
1 [&, foo]
```

Употребление `=` означает, что если имя захватываемой переменной не предварено символом `&`, то она будет захвачена по значению, а не по ссылке (C++14 5.1.2.14).

Кроме того, употребление любого из спецсимволов отдельно, означает, что захвачен `this`, тогда захватываемая переменная из контекста

класса или структуры не должна предваряться `this->`. В случае `=`, упоминание `this` – ошибка (C++14 5.1.2.8). Такая неоднозначность может несколько запутывать.

Важно помнить, что захват по ссылке это по умолчанию захват по константной ссылке и нужно ключевое слово `mutable` чтобы это изменить (C++14 5.1.2.5). При этом круглые скобки списка параметров если используется `mutable` опускать нельзя:

```

1  class Foo
2  {
3      int m_x;
4  public:
5      Foo () : m_x( 3 ) {}
6      void func ()
7      {
8          int x = 0;
9
10         /* by-copy x, error: x is readonly */
11         [x] { x += 3; } ();
12
13         /* by-copy x, ok */
14         [x] () mutable { x += 3; } ();
15
16         /* error: non-variable by-copy */
17         [m_x] () mutable { m_x += 3; } ();
18
19         /* by-ref x, ok */
20         [&x] () { x += 3; } ();
21
22         /* ok, x and this captured by ref */
23         [&] () { x += m_x; } ();
24
25         /* error: capture of non-variable m_x */
26         [&m_x] () mutable { std::cout << m_x; } ();
27
28         /* error: by-ref x, but this not captured */
29         [&x] () mutable { x += m_x; } ();
30
31         /* ok: by-ref x, this->mx used */
32         [=, &x] { x += m_x; } ();
33

```

```

34     /* ok: this->m_x used */
35     [=] () { std::cout << m_x; } ();
36 }
37 };

```

При захвате `m_x` выше на самом деле совершенно нет необходимости захватывать `this`, стандарт C++14 позволяет делать локальную копию при захвате:

```

1 /* error: non-variable by-copy */
2 [m_x] () mutable { m_x += 3; } ();
3
4 /* error: capture of non-variable m_x */
5 [&m_x] () mutable { std::cout << m_x; } ();
6
7 /* ok: local copy of m_x */
8 [captured_mx = m_x] () mutable { captured_mx += 3; } ();
9
10 /* ok: m_x field captured by reference */
11 [&captured_mx = m_x] () mutable { captured_mx += 3; } ();

```

Здесь амперсанд означает вовсе не взятие адреса. Взятие адреса, кстати, не сработает:

```

1 /* error: m_x field address by reference */
2 [&captured_mx = &m_x] () mutable { *captured_mx += 3; } ();

```

**Вопрос к студентам:** что здесь не так?

**Вопрос к студентам:** а если по значению?

```

1 /* ok: m_x field address by value */
2 [captured_mx = &m_x] () mutable { *captured_mx += 3; } ();

```

Возможность захватывать результат выражения в C++14 (generalized capture) даёт ещё более широкие возможности: захват по правой ссылке (через `std::move`), захват с конструированием объекта и так далее.

В захвате важно понимать, что захватываются только локальные нестатические переменные. Любая лямбда может свободно оперировать глобальным или статическим контекстом в её области видимости без всякого захвата (можно думать об этом как о захвате всегда по ссылке).

#### 4.7.4 Перегруженные лямбда-выражения

Могут ли лямбда функции быть перегружены? В наивном виде это не работает:

```
1 auto f = [](int i) { /* print forint */ };
2 auto f = [](double d) { /* print fordouble */ }; // No
```

Можно ли подойти к вопросу менее наивно? Да, если знать, что замыкание возвращает класс с правильно типизированным оператором круглые скобки. Тогда диспетчинг вызова можно возложить на класс, наследующий от всевозможных таких классов и таким образом содержащий в себе сколько угодно перегруженных круглых скобок (мультифункционтор):

```
1 template <class... F>
2 struct overload : F... {
3     overload(F... f) : F(f)... {}
4 };
```

И теперь:

```
1 template <class... F>
2 auto make_overload(F... f) {
3     return overload<F...>(f...);
4 }
```

Используется это как-то так:

```
1 auto f =
2     make_overload([](int i) { /* print forint */ },
3                     [](double d) { /* print fordbl */ });
4
5 f(3);
6 f(3.0);
```

Увы, это не работает. Перегруженные круглые скобки предка должны быть введены в контекст потомка `using`-объявлением (см. 3.7.3). Кроме того, здесь явно лишнее копирование в конструкторе.

Интуитивно хочется написать нечто вроде:

```
1 template <class... F>
2 struct overload : F... {
3     using overload<F...>::operator();
4     overload(F... f) : F(f)... {}
5 };
```

Но так делать нельзя.

Менее наивный подход предполагает раскрытие рекурсии:

```

1 template<class F, class... Fs>
2 struct overload : F, overload<Fs...>
3 {
4     using F::operator();
5     using overload<Fs...>::operator();
6
7     overload(F&& f, Fs&&... fs)
8         : F(std::move(f))
9         , overload<Fs...>(std::move(fs)...)
10    {}
11 };
12
13 template<class F>
14 struct overload<F> : F
15 {
16     using F::operator();
17
18     overload(F&& f)
19         : F(std::move(f))
20    {}
21 };

```

Очень интересно, что завершение рекурсии это анти-CRTP: тут не база параметризована наследником, а наоборот, наследника приходится специализировать последней базой. Это происходит потому, что в C++ нет перегрузки классов и простая строчка:

```

1 template<class F>
2 struct overload : F

```

в этом контексте не будет работать: у структуры `overload` два шаблонных параметра в точке объявления.

В этом примере пока не ясно от каких же типов наследует перегруженная структура. Пора решить этот вопрос.

#### 4.7.5 Типизация лямбда выражений: function type

Теперь пришло время разобраться с тем, как же типизированы  $\lambda$ -выражения.

$\lambda$ -выражение с пустой спецификацией захвата неявно приводится к обычному указателю на функцию в стиле C.

```
1 typedef int (*fptr_t)();
2 fptr_t fptr = [] { return 2; };
3 int x = fptr(); // ok
```

При необходимости использовать лямбду в шаблонном контексте, неявное приведение не работает для вывода типов. В этом случае используется унарный плюс для того, чтобы форсировать приведение.

```
1 template <typename R, typename A>
2 void foo (R, (*fptr)(A)) {
3     // .... do smth.
4 }
5
6 // error call:
7 foo ( [](double x) { return int(x); } );
8
9 // working call:
10 foo ( +[](double x) { return int(x); } );
```

Без унарного плюса здесь не сработает вывод типов, потому что для вывода типов шаблонной функцией неявное приведение не считается.

Пусть теперь есть необходимость создать  $\lambda$ -выражение с захватом контекста. Можно создать его использовав **auto** переменную:

```
1 int one_more_test(int x, int y) {
2     auto f1 = [&x, &y] { return x + y; };
3     return f1();
4 }
```

Но что если есть необходимость вернуть не результат вычисления  $\lambda$ -выражения, а его само, чтобы использовать его (со связанным контекстом) где-то ещё? Это можно сделать использовав **auto** как результат функции, но можно выразить эту же идею более явно через **std::function** (C++14 20.8)

```
1 function<int ()> harder_test(int x, int y)
```

```

2  {
3      return [&x, &y] { return x + y; };
4 }
```

Аналогично благодаря этому типу понятно как именно лямбда-функцию можно сделать рекурсивной

```

1 function<int (int)> factorial = [&] (int i)
2 {
3     return (i == 1) ? 1 : i * factorial(i - 1);
4 };
```

Немного странный в записи тип, которым параметризуется `std::function` уже неявно встречался в `typedef` и `using` выражениях. Так например:

```
1 typedef int factorial_t(int);
```

по факту содержит тот же тип `int (int)` для которого и вводится синоним (в данном случае `factorial_t`).

Но в чем тогда отличие `factorial_t` от `function<int (int)>?` Фундаментальное отличие в следующем: `function<int (int)>` понимает возможный захват контекста. При этом лямбда-функции с захватом разного контекста, будучи приведены к `std::function`, оказываются объектами одного типа.

```

1 int f(int a) { return -a; }
2
3 int main () {
4     int x = 0;
5
6     function<int(int)> fn1(f),
7         fn2 ([](int a) {return -a;}),
8         fn3 ([x](int a) {return x-a;});
9     cout << fn1.target_type().name() << endl
10        << fn2.target_type().name() << endl
11        << fn3.target_type().name() << endl;
12
13     .... etc ....
14 }
```

Этот пример выводит для GCC 5.3:

```
Z4mainEUiE_
Z4mainEUiE0_
```

Это потрясающий хак в систему типов C++, который получит свое развитие дальше.

Пока восхищение не прошло, его следует остудить: разумеется замыкания (closures), создаваемые лямбдами с разным контекстом имеют разные типы. Поэтому на само деле есть разница между двумя формами записи

```
1 auto closure = [x](int a) {return x-a;};
2 function<int(int)> func = [x](int a) {return x-a};
```

Замыкание имеет сразу правильный размер и хранится целиком на стеке. В то же время объект `function<int (int)>` имеет фиксированный размер и хранит само замыкание в динамической памяти. От этого его использование бывает гораздо менее эффективно.

Тем не менее, это не должно слишком охлаждать энтузиазм, потому что универсальность такого рода все равно восхитительна, даже если не совсем бесплатна.

Тип `std::function` переопределяет три ключевых оператора:

1. `operator=` – позволяет присвоить объекту класса новое значение
  2. `operator bool` – проверяет есть ли вообще значение:
- ```
1 std::function<void()> f1;
2 if (!f1) {
3     ... // here we are!
4 }
```
3. `operator()` – позволяет вызвать функцию.

**Вопрос к студентам:** в примере с имитацией перегрузки (см. 4.7.4), какими все-таки типами инстанцируются классы-предки перегруженных функций?

#### 4.7.6 Захват как способ уменьшения связности

Пусть необходимо разработать класс для отображения произвольной картинки на одном устройстве отображения. Логичная архитектура: от-

взять отображение от собственно картинки, сделав перерисовку каждого кадра callback-функцией для пользователя.

Проблема в том какую сигнатуру выбрать для этой функции:

```
1 enum class pollres {
2     PROCEED,
3     STOP
4 };
5
6 class ViewPort {
7     int width, height;
8     SDL_Surface *screen;
9     static ViewPort *v;
10
11    /* ??? */ callback;
12
13    ViewPort (int w, int h, /* ??? */ c)
14        : width (w), height (h), callback(c) {
15        if (SDL_Init(SDL_INIT_VIDEO) != 0)
16            throw runtime_error (SDL_GetError());
17
18        atexit(SDL_Quit);
19        screen = SDL_SetVideoMode(width + 1,
20            height + 1, 0, SDL_DOUBLEBUF);
21        if (screen == NULL)
22            throw runtime_error (SDL_GetError());
23    }
24
25 public:
26     pollres poll ();
27     void dump (const char *name);
28
29     static ViewPort *
30     QueryViewPort (int w, int h, /* ??? */ c) {
31        if (!v)
32            v = new ViewPort(w, h, c);
33        return v;
34    }
35
36    ~ViewPort() {
37        SDL_FreeSurface(screen);
```

```

38     }
39 };
40
41 void putpixel(SDL_Surface *surface,
42                 int x, int y, Uint32 color);
43 void fillwith (SDL_Surface *screen, Uint32 color);

```

Ясно, что функция рисования должна принимать `SDL_Surface`, чтобы рисовать на ней, но какие ещё параметры могут понадобиться пользователю? Правильный ответ – любые. В этом случае C-style это передавать туда `void *` и кастовать его к нужному типу (структуре обычно) каждый раз, когда вызывается callback.

В случае C++ можно использовать лямбды и захват контекста.

```
1 std::function<void(SDL_Surface*)> callback;
```

Если пользовательская функция реально берет больше параметров:

```

1 static void
2 draw_internal (SDL_Surface *s, int delta)
3 {
4 /* . . . . */
5 }
```

То ничего страшного:

```

1 int delta = 0;
2 auto draw_external = [&delta] (SDL_Surface *s) {
3     draw_internal (s, delta);
4 };
5
6 ViewPort *v = ViewPort::QueryViewPort (xsize, ysize,
7         draw_external);
8
9 for (delta = 4; delta < 10; ++delta) {
10    /* . . . . */
11 }
```

Таким образом развязывание зависимости позволяет сохранить сильную типизацию и избегать передачи чего-либо через `void *`

**Обсуждение в аудитории:** какие ещё методы можно предложить для решения той же архитектурной задачи?

#### 4.7.7 Провисание ссылок на захваченный контекст

При захвате контекста важно избегать возможностей провисания ссылок. Например объекты следующего класса:

```
1 struct Monitor {
2     ~Monitor () { cout << "destroyed" << endl; }
3 };
```

умеют сообщать о своем уничтожении. Теперь можно проанализировать типичное провисание ссылки

```
1 function<void()> f;
2
3 {
4     Monitor m;
5     auto k = [&mref = m] { cout << "may use m here" << endl; };
6     f = k;
7 }
8
9 f();
```

Если кто-то попытается вызвать сохранённую функцию `f` позже, чем закончится жизнь `x`, он получит все радости висячих ссылок (см. 2.5.2).

Естественный способ заблокироваться от возможности копирования замыкания это захватить в контекст некопируемое поле.

```
1 struct NC
2 {
3     NC()                  = default;
4     NC(NC const&)       = delete;
5     NC& operator=(NC const&) = delete;
6     NC(NC&&)           = default;
7 };
8
9 #define nocopy nocopy_value_ = NC()
```

Теперь простое добавление `nocopy` решает проблему:

```
1 function<void()> f;
2
3 {
4     Monitor m;
```

```

5   auto k = [nocopy, &mref = m] {
6       cout << "may use m here" << endl;
7   };
8   k ();
9   // f = k; // compilation error
10 }
11
12 // f(); // no way

```

Теперь такая лямбда просто не может покинуть свой контекст, а значит и захваченное в неё по ссылке не может зависнуть.

#### 4.7.8 Связывание аргументов

Самописная сохранялка из (4.6.2) равно как и каррирование из (4.7.2) являются не более чем частными случаями более общей техники, известной как связывание (binding) аргументов. Связывание это получение из чего-нибудь более частной функции путем частичной подстановки аргументов.

Первый, не слишком полезный пример (каррирование возвращается):

```

1 void f (int,int,int);
2 function<void()> f_call = bind (f, 1, 2, 3);
3 f_call (); //Equivalent to f (1,2,3)

```

Тип `function` даёт возможность правильно типизировать связанную функцию, а вариантов упомянутого чего-нибудь язык предлагает массу.

Иллюстративный пример:

```

1 struct Foo {
2     Foo(int num) : num_(num) {}
3     void print_add(int i) const {
4         std::cout << num_+i << std::endl;
5     }
6     int num_;
7 };

```

Использование:

```

1 const Foo foo(314159);
2 using std::placeholders::_1;
3 function<void(int)> f_add_display2 =

```

```
4     bind( &Foo::print_add, foo, _1);
```

Функция `f_add_display2` это функция, которая получилась из метода класса, связанного со своим объектом и первым аргументом этой функции как заместителем. Термин `placeholder` (заместитель) отображает семантику происходящего: даёт способ заместить аргумент в вызове. Цифра в заместителе это номер аргумента при вызове который будет подставлен в эту позицию при связывании.

```
1 function<void(int,int,int)> f_call = bind (f, _1, _2, _3);
2 f_call (1, 2, 3); //Same as f (1, 2, 3)
3
4 function<void(int,int,int)> reordered_call =
5     bind (f, _3, _2, _1);
6
7 reordered_call (3, 2, 1); // Same as f( 1 , 2 , 3 );
```

Тут стоит попрактиковаться. Очередной пример с выводом на экран.

```
1 void f(int n1, int n2, int n3, const int& n4, int n5) {
2     cout << n1 << ',' << n2 << ',' << n3 << ','
3         << n4 << ',' << n5 << endl;
4 }
```

Ну и чтобы было интереснее, вызов можно несколько запутать:

```
1 using namespace std::placeholders; // for _1, _2, ...
2 int n = 7;
3 auto f1 = std::bind(f, _2, _1, 42, std::cref(n), n);
4 n = 10;
5 f1(1, 2, 1001);
```

**Вопрос к студентам:** что будет на экране?

Теперь можно видеть, что несколько загадочный по назначению `cref` из (3.5.3) нашёл свое применение. С его помощью можно сделать константную ссылку для передачи в `bind`.

Если класс является функтором (содержит переопределенный `operator()`) то можно связаться не с его методом а с ним самим, почти так же просто как с функцией:

```
1 uniform_int_distribution<> d(0, 10);
2 default_random_engine e;
3 function<int()> rnd = bind(d, e);
```

Этот пример немного забегает вперед, так как случайные числа ещё не рассматривались. Тем не менее, всё, в общем, ясно.

**Вопрос к студентам:** кто догадается как использовать `rnd`?

#### 4.7.9 Variadic lambdas и монады

*A parser for things*

*Is a function from strings*

*To lists of pairs*

*Of things and strings*

– Graham Hutton

До сих пор монады были сладким поводом задрать нос для всяких там программистов на языке Haskell. Между тем, с введением элементов функционального программирования, C++ стал справляться не хуже.

Что такое монада? На этот вопрос есть математически-корректный, но бесполезный ответ: “моноид на категории эндофункций” (общий смех). К счастью, мир проще. Программист на C++ может воспринимать монаду как перегруженный оператор точка с запятой.

Простейшая монада `State` можно считать уже есть благодаря императивной природе языка:

```
1 x = foo ();
2 z = bar ();
3 y = buz (x, z);
```

**Вопрос к студентам:** чем это отличается от:

```
1 y = buz (foo(), bar());
```

Также в языке есть монада `Mayube` благодаря логике сокращенным выполнением:

```
1 (x = foo ()) &&
2 (z = bar ()) &&
3 (y = buz (x, z));
```

Здесь каждый следующий шаг возможен только если предыдущий вернул не ноль.

Ниже речь пойдет о третьей интересной монаде List – операциях над списками произвольных объектов. Правда понадобится ещё немного синтаксического сахара, поэтому сначала следует поговорить про вариабельные лямбды.

Поскольку лямбды внутри являются шаблонами, они наверное могут быть и вариабельными шаблонами? Так точно – отвечают нам создатели языка. В стандарт C++14 введены variadic lambdas – расширение рассмотренных выше generic lambdas у которых обобщенные параметры ведут себя как если бы они были вариабельными шаблонами.

Изложение в этом разделе в целом следует <http://cpptruths.blogspot.ru/2014/08/fun-with-lambdas-c14-style-part-3.html> (TODO: включить в список литературы?)

Сначала конструкторы и голова/хвост списка:

```

1 auto List = [](auto ... xs) {
2     return [=](auto access) {
3         return access(xs...);
4     };
5 };
6
7 auto head = [](auto xs) {
8     return xs([](auto first, auto ... rest) {
9         return first;
10    });
11 };
12
13 auto tail = [](auto xs) {
14     return xs([](auto first, auto ... rest) {
15         return List(rest...);
16    });
17 };
18
19 auto length = [](auto xs) {
20     return xs([](auto ... z) { return sizeof...(z); });
21 };

```

Это позволяет создавать списки “как в SICP” и делать над ними привычные операции:

```

1 auto three = length(List(1, '2', "3"));
2 cout << three << endl;

```

**Вопрос к студентам:** какого типа аргумент `xs` в функции `length`?

Для приведенного выше списка можно написать, например, функтор `fmap`, у которого на Haskell была бы сигнатура вроде:

```
fmap: (a -> b) -> list[a] -> list[b]
```

Для C++ он будет выглядеть ничуть не хуже:

```
1 auto fmap = [](auto func) {
2     return [func] (auto alist) {
3         return alist(
4             [func](auto ... xs)
5             {
6                 return List(func(xs)...);
7             }
8         );
9     };
10};
```

Например:

```
1 auto twice = [](auto i) { return 2*i; };
2 auto print = [](auto i) { std::cout << i << " "; return i; };
3 auto l1 = List(1, 2, 3, 4);
4 auto l2 = fmap(twice)(l1);
5 auto l3 = fmap(print)(l2);
```

**Вопрос к студентам:** в каком порядке будут выведены элементы списка?

**Домашняя наработка:** сделать `fmap` с упорядоченными побочными эффектами.

Критичной для монады является операция `flatmap : (a -> list[b]) -> list[a] -> list[b]` её для простоты можно разложить (для списков) на конкатенацию:

```
1 auto concat = [](auto l1, auto l2) {
2     auto access1 = [=](auto... p) {
3         auto access2 = [=](auto... q) {
4             return List(p..., q...);
5         };
6         return l2(access2);
```

```

7     };
8     return l1(access1);
9 }

```

И флаттенинг:

```

1 template <class Func>
2 auto flatten(Func)
3 {
4     return List();
5 }
6
7 template <class Func, class A, class... B>
8 auto flatten(Func f, A a, B... b)
9 {
10    return concat(f(a), flatten(f, b...));
11 }

```

Вместе собирается как функция высшего порядка:

```

1 auto flatmap = [](auto func) {
2     return [func](auto alist) {
3         return alist(
4             [func](auto... xs) {
5                 return flatten(func, xs...);
6             });
7     };
8 };

```

Теперь можно сделать много интересных вещей со списком:

```

1 auto pair = [](auto i) { return List(-i, i); };
2 auto count = [](auto... a) { return List(sizeof...(a)); };
3
4 auto l4 = List(1, 2, 3);
5 auto l5 = flatmap(pair)(l4);
6 auto l6 = fmap(print)(l5);

```

По сути fmap и flatmap это уже монадические операции. Осталось привести их к стандартному виду операторов > и >=:

```

1 template <class LIST, class Func>
2 auto operator > (LIST l, Func f)
3 {

```

```
4     return fmap(f)(l);
5 }
6
7 template <class LIST, class Func>
8 auto operator >= (LIST l, Func f)
9 {
10    return flatmap(f)(l);
11 }
```

И вот, монада List получилась не хуже, чем в Haskell.

```
1 auto l7 =
2 List(1, 2, 3) >= pair > print;
```

**Домашняя наработка:** монадический подъём вычислений определяется как:  $liftM2 :: Monad m \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow m a \rightarrow m b \rightarrow m c$ . Ваша задача сделать версию `liftM2` для монады `List` из этого примера.

В эпиграф этого подраздела вынесена цитата из <http://eprints.nottingham.ac.uk/223/1/pearl.pdf> где рассматривается монадический подход к синтаксическому разбору программ.

## 4.8 Правила инстанцирования и SFINAE

К этому моменту уже несколько раз было употреблено слово “инстанцирование”. Интуитивно оно понятно, но оно имеет разный смысл для обычных классов (там инстанцирование класса это создание его объекта) и для шаблонов. Настало время дать этому слову более точное определение.

**Шаблонным инстанцированием** называется процесс порождения типов и функций из обобщённых шаблонных определений.

Инстанцирование может также включать в себя вывод типов для шаблонных параметров (type inference), изобретение типов обобщенных лямбда функций (type invention) и в конце всегда включает подстановку шаблонных параметров (type substitution).

Порождение из обобщённого шаблона конкретного класса (конкретной функции etc....) это всегда немного волшебство. Некоторым кажется, что не стоит глубоко разбираться в механизмах этого волшебства, потому что это внутреннее дело компилятора. Не может быть ничего вреднее, чем это заблуждение! Тонкое понимание механизмов инстанцирования позволяет прикоснуться к миру удивительного и невероятного (и в том числе – в прикладном коде).

### 4.8.1 Инстанцирование шаблонов

Ключевой точкой инстанцирования является подстановка типов. Именно корректная подстановка типов определяет в конечном итоге сигнатуру порожденной из шаблона функции.

Разумеется, функция или класс, порождённые из обобщённого определения должны где-то физически быть в коде. Это важно потому, что некоторые вещи в языке зависят от местоположения определения (такие как специализация или возможность использовать полный тип). Здесь важно помнить: шаблоны классов инстанцируются до своего первого упоминания, а шаблоны функций – после.

```

1 template<typename T, int N> struct Stars {
2     using t = typename Stars<T, N-1>::t*;
3 };
4
5 template<typename T> struct Stars<T, 0> {
6     using t = T;

```

```
7 };
```

Где-то в коде эта конструкция может быть использована, например, так.

```
1 using ipptr_t = typename Stars<int, 2>::t;
```

Можно проследить цепочку инстанцирований

```
1 class Stars<int, 0> { using t = int; };
2 class Stars<int, 1> { using t = typename Stars<int, 0>::t*; };
3 class Stars<int, 2> { using t = typename Stars<int, 1>::t*; };
4 using ipptr_t = typename Stars<int, 2>::t; // yields int**
```

Видно, что если бы невидимая глазу точка, куда компилятор помещает настоящее определение конкретного экземпляра обобщённого типа была после использования этого типа, то всё бы существенно запуталось.

Ещё более красивым примером (хотя и несколько синтетическим), сочетающим инстанцирование и классов и функций является пример, который я называю **танец с функциями**.

```
1 template<typename T> void proceed(T);
2
3 template<typename T> struct Dancing {
4     void tearup() { proceed(0); }
5     void finalize() { hic salta }
6 };
7
8 template<typename T> void proceed(T t) {
9     Dancing<T> a;
10    a.finalize();
11 }
12
13 int main() {
14     Dancing<int> a;
15     a.tearup();
16 }
```

Сначала следует окинуть его внимательным взглядом и понять что происходит и как это распутывать. Итак, `main` инстанцирует `Dancing<int>`. Внутри метода `tearup`, вывод типов инстанцирует `proceed<int>`, которой, в свою очередь, требуется снова `Dancing<int>`. Как теперь выкрутится компилятор?

```
1 template<typename T> void proceed(T);  
2  
3 template<typename T> struct Dancing {  
4     void tearup() { proceed(0); }  
5     void finalize() { hic salta }  
6 };  
7  
8 struct Dancing<int> {  
9     void tearup();  
10    void finalize();  
11};  
12  
13 int main() {  
14     Dancing<int> a;  
15     a.tearup();  
16 }  
17  
18 void Dancing<int>::tearup () { proceed(0); }  
19  
20 void proceed<int> (int) {  
21     Dancing<int> a;  
22     a.finalize();  
23 }  
24  
25 void Dancing<int>::finalize () { hic salta }
```

Этот код в целом понятен. Ключевой момент: компилятор делает инстанцирование для методов как для функций (то есть где-то **после** их реального упоминания).

В книгах по шаблонам, таких как [11] традиционно много места отводится тому, что называется “Separation model”. Это модель инстанцирования, когда код шаблона находится в одной единице трансляции, а инстанцирование происходит в другой. Начиная с C++11 экспорт шаблонов был запрещён, а Separation model объявлена устаревшей. Поэтому везде далее можно полагать, что код шаблона размещается и инстанцирование происходит в одной единице трансляции.

### 4.8.2 Достоинства ленивости

С и C++ – это языки с энергичной (с некоторыми исключениями) редукцией выражений при вычислениях, поэтому изучающие их (если у них нет предварительного опыта в таких языках как Lisp или Haskell) часто даже не подозревают, что может быть как-то иначе.

```

1 int foo (int x, int y)
2 {
3     return (x > 3) ? 0 : y;
4 }
5 /* ... */
6 foo (a + 3, b + 2);

```

Ленивые и энергичные вычисления для этого кода проиллюстрированы на рисунке

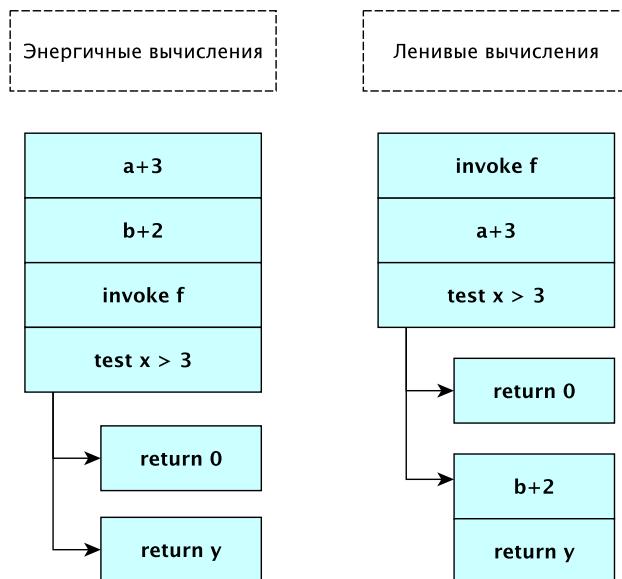


Рис. 4.1: Ленивые и энергичные вычисления

Видно, что при ленивых вычислениях, реальное вычисление результата выражения откладывается насколько это возможно. Это создает свои проблемы: скажем передавать в функцию такое недосчитанное выражение гораздо сложнее, чем передать посчитанное заранее число. Поэтому ленивые вычисления в C++ встречаются только в нескольких контекстах:

1. В вычислениях с длинными логическими операциями: `&&` и `||`. Из-за ленивого порядка их применения, легальны выражения вида:

```
1 if (p && p->next) { /* ... */ }
```

Если бы порядок вычисления был энергичным, то при нулевом указателе второе условие взорвалось.

2. В инстанцировании шаблонов

3. В исполнении константно-выраженных функций, подробнее см. (4.11.1).

Итак шаблонное инстанцирование обязано работать лениво. Это означает, что проверены на корректность будут только те части шаблона, которые действительно были использованы в коде.

Пример вроде бы нарушает стандарт

```
1 template <int N>
2 struct Danger {
3     typedef char block[N];
4 };
5
6 template <typename T, int N>
7 struct Tricky {
8     void test_lazyness() {
9         Danger<N> no_boom_yet;
10    }
11 };
12
13 int main() {
14     Tricky<int, -2> ok;
15 }
```

Несмотря на то, что инстанцирование определяет `typedef` с некорректным параметром размера массива, это не является ошибкой, поскольку реально этот зависимый тип нигде не был использован. Ленивость вынуждает компилятор откладывать инстанцирование любой части шаблона так долго, как это возможно.

### 4.8.3 Идиома SFINAE

*I can accept failure, everyone fails at something.*

*But I can't accept not trying.*

— Michael Jordan

Одной из важнейших идиом в правилах инстанцирования является SFINAE, что означает **substitution failure is not an error**. Она означает, что неудача при подстановке шаблонных параметров сама по себе не является ошибкой.

То, что я перевожу как “неудача” подстановки (substitution failure) это на самом деле термин, означающий скорее невозможность, то есть в слове failure нет оттенка недетерминированности, который, увы, есть в слове неудача. При чтении это надо понимать.

Формально SFINAE определяется так: если в результате подстановки в **непосредственном контексте** класса (функции, алиаса, переменной) возникает **невалидная конструкция**, эта подстановка неуспешна, но не ошибочна. Сочетание этих двух условий порождает **SFINAE-case**.

Неудачную подстановку можно продемонстрировать на примере уже полюбившейся всем функции максимума:

```
1 template <typename T> inline const T
2 max (const T a, const T b)
3 {
4     return (a > b) ? a : b;
5 }
```

Семантически здесь указано, что оба аргумента имеют одинаковый тип. Таким образом, подстановка, в которой для них будут выведены разные типы, будет неудачной.

```
1 int g = max (1, 1.0);
```

Что произойдёт? Благодаря принципу SFINAE, наткнувшись на неудачную подстановку, компилятор не будет паниковать и выдавать сообщения об ошибках.

Возможно где-то определён более подходящий шаблон, например:

```
1 template <typename T, typename U> inline const T
2 max (const T a, const U b) { ... }
```

и подстановка в него завершится успехом.

Если нет, то сообщение об ошибке будет выглядеть довольно очевидно:

```
error: no matching function for call to ‘max(int, double)’
note: candidate is:
note: template<class T> const T max(T, T)
note: template argument deduction/substitution failed:
note: deduced conflicting types for parameter
      ‘T’ (‘int’ and ‘double’)
```

Ещё один пример неудачной подстановки – порождение кода, не соответствующего типу данных. Например ниже у выведенного типа `int*` явно нет зависимого подтипа `ElementT`.

```
1 template<typename T>
2 typename T::ElementT at (T const& a, int i) {
3     return a[i];
4 }
5
6 int
7 f (int *p) {
8     int x = at(p, 7); /* boom! */
9     return x;
10 }
```

Но само по себе это тоже не ошибка. С другой стороны, здесь конечно сложно предложить как следует переписать код, чтобы специализировать функцию для правильной подстановки.

Очевидный вариант:

```
1 template<typename T, typename U>
2 U at (T const& a, int i) {
3     return a[i];
4 }
```

Не работает, потому что компилятору неоткуда вывести тип `U`.

Более интересный вариант использует новый стандарт и не требует наличия зависимых типов:

```
1 template<typename T>
```

```

2 auto at (T const& a, int i) -> decltype(a[i]) {
3     return a[i];
4 }
```

Здесь подстановка пройдет удачно. Интересно, что стандарт не допускает перегрузку функции по возвращаемому значению, но допускает перегрузку шаблонной функции по возвращаемому значению **и типу**.

Не любая ошибка инстанцирования это SFNAE-case. Подстановка должна быть в непосредственном контексте шаблонного объявления, это иногда нетривиально.

```

1 int negate (int i) { return -i; }
2
3 template <typename T> T negate(const T& t) {
4     typename T::value_type n = -t();
5     // тутиспользуем n
6 }
7
8 negate(2.0); // ошибка
```

Здесь в контексте сигнатуры и шаблонных параметров нет никакой невалидности. Невалидность в теле не является SFNAE, это ошибка второй фазы трансляции.

Однако же, если переписать функцию `negate` так, чтобы сделать из неё SFNAE-case, например введя неудачный подтип в возвращаемое значение, как показано ниже, тогда это будет ошибкой подстановки и выберется перегрузка.

```

1 template <typename T> T::value_type negate(const T& t) {
2     typename T::value_type n = -t();
3     // тутиспользуем n
4 }
```

Такие случаи нужно чувствовать и осознавать, что они могут втсречаться.

SFINAE является очень мощной техникой, позволяющей делать интересные вещи. Например определить на этапе компиляции имеет ли некий тип вложенный зависимый тип с определённым именем.

```

1 template <typename T> struct has_TYPEDEF_foobar {
2     typedef char yes[1];
3     typedef char no[2];
```

```

4
5     template <typename C>
6     static yes& test(typename C::foobar*) ;
7
8     template <typename>
9     static no& test(...);
10
11    static const bool value =
12        sizeof(test<T>(0)) == sizeof(yes);
13 };
14
15 struct foo {
16     typedef float foobar;
17 };
18
19 struct bar {
20 };
21
22 int main() {
23     cout << "foo: " << has_TYPEDEF_foobar<foo>::value
24         << "\nbar: " << has_TYPEDEF_foobar<bar>::value
25         << endl;
26 }
```

Этот код выведет результат `foo: 1, bar: 0`, верно определив наличие зависимого типа `foo::foobar` в `foo` и его отсутствие в `bar`. Эта техника может казаться с первого взгляда не очевидной. Я рекомендую изучить идею, пока она не станет ясна: понимание SFNAE это ключ к пониманию всего метапрограммирования.

**Домашняя наработка:** Попробуйте в таком же стиле проверить наличие метода `foobar`, возвращающего обязательно `float` и не берущего ни одного аргумента. Тривиальная замена:

```

1 template <typename C> static auto test(void*) ->
2     decltype(float {declval<C>().foobar()}, yes{});
```

До какой-то степени работает, но не ловит случаи когда возвращается `float&` и т.п.

Всё это слишком запутывается. Я называю такое SFNAE “партизанским”, потому что приходится блуждать в страшных зимних лесах без еды годами с оружием. Поэтому люди искали более систематические

подходы.

И нашли их.

## 4.9 Систематическое SFINAE

Проблемы с “партизанским” SFINAE во многом были вызваны тем, что область его применения была очерчена недостаточно исчерпывающе. Ясно, что возможны какие-то трюки, но какие трюки возможны, а какие нет, и, что важнее **почему** они вообще возможны?

### 4.9.1 Истина и ложь

Ключевой шаг в систематизации SFINAE это понимание того, что в языке кроме “пространства типов” и “пространства значений” есть также “пространство характеристик валидности” или SFINAE-space и при любой работе в SFINAE-контекстах мы отображаем одно на другое.

Базовое отображение типов на значения и обратно даётся удобным классом (часть стандарта C++14) `integral_constant`

```

1 template <typename T, T v>
2 struct integral_constant {
3     static const T value = v;
4     typedef T value_type;
5     typedef integral_constant type;
6     operator value_type() const { return value; }
7 };

```

Функции, работающие в SFINAE контекстах, должны как-то сообщать ответы “да” и “нет” на вопросы “является ли это тем-то и тем-то?”, скажем: “является ли тип аргумента указателем на функцию-член класса X с такой-то сигнатурой?”. Чтобы закодировать ответы, можно использовать простейшие из таких констант: интегральные булевые константы.

```

1 using true_type = integral_constant<bool, true>;
2 using false_type = integral_constant<bool, false>;

```

И они же позволяют отображение из type-space на sfinae-space

```

1 template<typename T, typename U>
2 struct is_same : false_type {};
3
4 template<typename T>
5 struct is_same<T, T> : true_type {};

```

Теперь благодаря SFINAE, будет работать

```

1 assert (is_same<int, int>::value &&
2     !is_same<char, int>::value);

```

Пользоваться каждый раз `xxx::value` не всегда удобно. Чуть более удобно – сделать алиас в виде шаблонной переменной.

```

1 template<typename T, typename U>
2 bool is_same_v = is_same<T, U>::value;

```

Теперь будет работать как полная, так и сокращённая версия

```

1 assert (is_same_v<int, int> &&
2     !is_same_v<char, int>);

```

Интересно, что эта проверка делает фактически обратное отображение из SFINAE-space (валидности и невалидности) на значения.

#### 4.9.2 Определители и модификаторы

Естественное применение SFINAE, как следует из прошлого примера это определители и модификаторы типов. Задача определителя – сказать что-то о типе (есть ли в нём ссылки и указатели, константен ли он или волатилен и т.п.), задача модификатора – сделать что-нибудь с типом (убрать ссылку, добавить указатель, сменить волатильность на константность)

Пример стандартного определителя: является ли тип ссылкой

```

1 template <typename T> struct is_reference : false_type {};
2 template <typename T> struct is_reference<T&> : true_type {};
3 template <typename T> struct is_reference<T&&> : true_type {};

```

Пример стандартного модификатора: убираем ссылку с типа

```

1 template <typename T>
2 struct remove_reference { using type = T; };
3
4 template <typename T>
5 struct remove_reference<T&> { using type = T; };
6
7 template <typename T>
8 struct remove_reference<T&&> { using type = T; };

```

Для модификатора полезен алиас (см. 4.2.9)

```

1 template <typename T>
2 using remove_reference_t = typename remove_reference<T>::type;

```

Я называю такие конструкции “триадами” и очень часто именно в три строчки записывается SFINAE определитель или модификатор.

К счастью, некоторые конструкции являются очень простыми. Настолько простыми, что, казалось бы, не требуют даже триад. Например добавление обычной левой ссылки

```

1 // this code is terribly wrong
2 template <typename T> struct add_lref { using type = T&; };

```

Ну что может пойти не так при добавлении левой ссылки?

Но увы. В таком виде добавление левой ссылки не будет работать для `void`. Причина банальная – `void&` это некорректно. В некорректном случае, как уже было сказано выше, хорошая идея это оставить всё как есть. Можно пойти путём инкрементальных улучшений и предусмотреть случай для `void`.

```

1 template <typename T> struct add_lref { using type = T&; };
2 template <> struct add_lref<void> { using type = void; };
3
4 template <>
5 struct add_lref<const void> { using type = const void; };
6
7 template <>
8 struct add_lref<volatile void> { using type = volatile void; };
9
10 template <>
11 struct add_lref<const volatile void> {
12     using type = const volatile void;
13 };

```

Красота пропала, но проблема не в этом. Проблема в том, что, а почему мы вообще уверены, что только `cv-void` так себя ведёт? Кажется, мысль тут изначально пошла по неверному пути. Верный путь лежит через SFINAE: необходимо выразить **идею** того, что либо мы добавляем ссылку если можем, либо оставляем всё как есть если не можем.

```

1 template <typename T, typename Enable>
2 struct ALRImpl { using type = T; }
3

```

```

4 template <typename T>
5 struct ALRImpl <T, remove_reference_t<T&>> {
6     using type = T&;
7 }
8
9 template <typename T>
10 struct add_lref : ALRImpl <T, remove_reference_t<T>> {}

```

Эта изящная триада и многие другие приведена в частности в докладе Arthur O'Dwyer на CppCon 2017 (TODO: добавить в список литературы?)

Решение красиво, но тяжеловесно из-за того, что полагается на поведение `remove_reference_t`.

Сделать его куда более прозрачным можно с помощью шаблона `void_t` (стандартизирован в 2017 году).

```
1 template <class ...> using void_t = void;
```

Это по сути отображение пачки типов на `Enabled`, если каждый из них `Enabled`. С использованием этого улучшения, искомый модификатор будет выражен куда яснее.

```

1 template <typename T, typename Enable>
2 struct ALRImpl { using type = T; }
3
4 template <typename T>
5 struct ALRImpl <T, void_t<T&>> { using type = T&; }
6
7 template <typename T>
8 struct add_lref : ALRImpl <T, void_t<>> {}

```

Здесь убрано всё лишнее и можно ли сделать лучше – не очевидно.

Системный подход к SFINAE делает даже тяжеловесный определитель наличия зависимого типа с данным именем (`has_TYPEDEF_foobar` из 4.8.3) весьма простым и понятным.

```

1 template <typename, typename = void_t<>>
2 struct has_TYPEDEF_foobar : false_type {};
3
4 template <typename T>
5 struct has_TYPEDEF_foobar <T,
6     void_t<typename T::foobar>> : true_type {};

```

Основные преимущества системного подхода: ясное и корректное выражение мысли в коде. Язык продолжает развитие в этом направлении. Например в C++17 приведённый выше код можно переписать ещё проще, используя появившийся там `std::is_detected`

```
1 template <typename T>
2 using has_TYPEDEF_foobar_t = decltype(T::foobar);
3
4 cout << is_detected<has_TYPEDEF_foobar_t, foo>::value << endl;
```

Тем не менее, много до сих пор остаётся творческим процессом.

### 4.9.3 Вариабельные шаблоны для списков типов

Типичная задача на конструирование определителей типов: сконструировать `and_t <T, U>`, который является `true_type` только если и `T` и `U` являются `true_type`, иначе является `false_type`.

Решение к этому моменту должно быть уже почти очевидно:

```
1 template <typename T, typename U>
2 struct and_t : false_type {};
3
4 template <>
5 struct and_t <true_type, true_type> : true_type {};
```

Но что если необходимо написать чуть более сложный определитель: вычислить конъюнкцию целого списка типов? Или, скажем, установить равенство данного элемента одному из списка?

Здесь на помощь приходят вариабельные шаблоны.

```
1 template<typename T, typename... List>
2 struct is_one_of;
3
4 template<typename T>
5 struct is_one_of<T> : false_type {};
6
7 template<typename T, typename... Tail>
8 struct is_one_of<T, T, Tail...> : true_type {};
```

Самое сложное и изящное это рекурсивный вызов:

```
1 template<typename T, typename Head, typename... Tail>
```

```

2 struct is_one_of<T, Head, Tail...> :
3     is_one_of<Head, Tail...> {};

```

Таким образом вариабельные шаблоны позволяют рекурсию по списку типов гораздо естественней, чем это позволяли делать обычные шаблоны. Большое количество кода в известной книге [15] теперь может быть переписано гораздо проще.

**Вопрос к студентам:** Дан template parameter pack ...Ts, где каждый тип может быть либо `true_type` или `false_type`. Как узнать, являются ли они все `true_type`?

Очевиден простой рекурсивный ответ:

```

1 template <typename ...Ts>
2 struct all_true;
3
4 template <typename H, typename ...Ts>
5 struct all_true<H, Ts...> :
6     and_t<is_same<true_type, H>,
7         all_true<Ts...>> {};
8
9 template <>
10 struct all_true<> : true_type {};

```

Можно ли сделать изящней, не инстанцируя  $O(N)$  типов?

Часть из использованных здесь определителей есть в стандарте, другие можно сконструировать по мере необходимости (хотя иногда это бывает нетривиально).

Можно запомнить (это примерно столь же полезная для запоминания информация как первый 21 знак числа пи), что все что угодно, что встречается в корректной программе на C++14 может быть отнесено к одному из 14 базовых классов traits и только к нему одному.

```

1 template <class T> struct is_void;
2 template <class T> struct is_null_pointer;
3 template <class T> struct is_integral;
4 template <class T> struct is_floating_point;
5 template <class T> struct is_array;
6 template <class T> struct is_pointer;
7 template <class T> struct is_lvalue_reference;
8 template <class T> struct is_rvalue_reference;
9 template <class T> struct is_member_object_pointer;

```

```

10 template <class T> struct is_member_function_pointer;
11 template <class T> struct is_enum;
12 template <class T> struct is_union;
13 template <class T> struct is_class;
14 template <class T> struct is_function;

```

И кстати 14 это первые две цифры номера стандарта. Совпадение?  
Не думаю.

#### 4.9.4 Статический асерт

Некоторые утверждения можно проверить на этапе компиляции. Конечно, на этапе исполнения их, обычно, тоже можно проверить.

```

1 int foo () {
2     assert (sizeof(int) == 4);
3     // more code here
4 }

```

Допустимо, но выглядит странно. И потом: неужели это нужно делать в каждой функции?

Идея: написать в глобальной области видимости

```
1 CT_ASSERT (sizeof(int) == 4);
```

Удивительно, но это возможно даже на С. Первое, что приходит на ум, это нечто вроде этого:

```
1 #define CT_ASSERT(pred) switch(0){case 0:case pred:;}
```

Или вот этого:

```
1 #define CT_ASSERT(pred) do {int arr[pred ? 1 : -1];} while(0);
```

Оба способа известны чуть ли не с восьмидесятых и оба в целом негодные, потому что не годятся для размещения вне функций.

Годный способ реализации на языке С чуть более сложен (пример взят с <https://stackoverflow.com/questions/807244/c-compiler-asserts-how-to-implement>)

```

1 #define CT_ASSERT(predicate) \
2     impl_CASSERT_LINE(predicate, __LINE__, __COUNTER__)
3 #define impl_PASTE(a,b) a##b
4 #define impl_CASSERT_LINE(predicate, line, cnt) \

```

```
5  typedef char impl_PASTE( \
6      assertion_failed_##cnt##_,line)[2*!!(predicate)-1];
```

Он построен на том же контроле массивов отрицательной длины, но благодаря использованию **typedef** подходит для глобальных упоминаний. Но нельзя не заметить, что этот способ какой-то мрачный...

Андрей Александреску [15] в 2001 году предложил красивый и специфичный для C++ способ решения проблемы статического ассерта.

```
1 template <bool cond> struct CT_ASSERT;
2 template <> struct CT_ASSERT<true> {};
```

Эти две строчки поражают воображение своей элегантностью. И в целом это работает, но тут есть проблемы с выдачей сообщения (большинство ошибок в шаблонах удивительно нечитаемы), писать в коде надо треугольные скобки, а не круглые и, главное, приходится придумывать имя для каждого асверта.

```
1 CT_ASSERT<sizeof(int) == 4> myassert1;
```

Это неудобно, так что с 2011 года на уровне языка ввели синтаксический сахар.

- Для языка C: **\_Static\_assert (cond, message)**
- Для языка C++: **static\_assert (cond, message)**

Статические асверты бесплатны и служат частью документации кода.

```
1 #include "mylib.h"
2 static_assert (MyLib::Version > 2, "Old Mylib");
3 static_assert (sizeof(int) == 4, "Incompatible environment");
4 // .... etc
```

Если что-то можно проверить статически, лучше проверить это статически.

Для нас интересно, что статический асверт пришлось вводить на уровне языка именно потому что SFINAE-путь его организации, хоть и существовал, оказался неудобным.

Тем не менее, для многих других вещей этот путь не только удобный, а, пожалуй и единственный.

#### 4.9.5 Условный тип и условный Enable

Рассмотренный ранее статический асерт отображает выражения на валидность. Можно ли отображать типы в типы либо невалидные характеристики в зависимости от выражений? Оказывается, что да.

Сначала простое, управляемое выражением, отображение типов.

```

1 template <bool C, typename T, typename F>
2 struct conditional { using type = T; }
3
4 template <typename T, typename F>
5 struct conditional<false, T, F> { using type = F; }
6
7 template <bool C, typename T, typename F>
8 using conditional_t = typename conditional<C, T, F>::type;
```

Эта триада называется условным типом. Она отображает значение C (от англ. condition) как `{true, false} -> {T, F}`.

Теперь можно проделать интересную операцию: выкинуть из условного типа всё, касающееся F. В этом случае получается отображение `{true, false} -> {T, Disabled}`

```

1 template <bool B, typename T = void>
2 struct enable_if { using type = T; }
3
4 template <typename T = void>
5 struct enable_if<false, T> { }
6
7 template <bool B, typename T = void>
8 using enable_if_t = typename enable_if<B, T>::type;
```

Полученный паттерн – один из самых известных и один из самых полезных изо всех. Главное в нём то, что он позволяет выкидывать (SFINAE-out) нежеланные инстанциации шаблонов.

```

1 template <typename T,
2         typename = enable_if_t<(sizeof(T) > 4)>>
3 void foo (T x) { do something with x }
4
5 foo('c'); // FAIL
```

До некоторой степени, `enable_if` штука коварная. Например, очевидная задача: написать в пару приведённой выше функции, функцию,

которая работает для (`sizeof(T) <= 4`) не так уж и просто.

Очевидная идея не работает.

```

1 template <typename T,
2         typename = enable_if_t<(sizeof(T) > 4)>>
3 void foo (T x) { do something with x }
4
5 template <typename T,
6         typename = enable_if_t<(sizeof(T) <= 4)>>
7 void foo (T x) { do something with x }
8
9 foo('c'); // overload failure

```

Увы, до разрешения подстановки и вообще до SFINAE здесь не доходит. С точки зрения компилятора, написана одна и та же функция и выбрать перегрузку не представляется возможным.

Можно, конечно, выкрутиться с dummy аргументом

```

1 template <typename T,
2         typename = enable_if_t<(sizeof(T) <= 4)>>
3 void foo (T x, int dummy = 0) { do something with x }

```

Это работает, но совсем непонятно откуда такая асимметрия на равном месте.

Одна из возможных правильных идей: разруливать этот контекст не шаблонным параметром, а перегруженным типом возвращаемого значения.

```

1 template <typename T>
2 enable_if_t<(sizeof(T) > 4), void>
3 foo (T x) { do something with x }
4
5 template <typename T>
6 enable_if_t<(sizeof(T) <= 4), void>
7 foo (T x) { do something with x }
8
9 foo('c'); // ok

```

Так всё работает.

**Вопрос к студентам:** Является ли хорошей идея пожертвовать типом аргумента?

```
1 template <typename T> void
2 foo (enable_if_t<(sizeof(T) > 4)>, T> x) {
3 do something with x
4 }
```

Это тоже работает, но сделали ли бы вы так?

Красивое и симметричное решение для этой задачи также получается если рулить указателями а не типами в качестве шаблонных параметров.

```
1 template <typename T,
2         enable_if_t <(sizeof(T) > 4)>*> = nullptr>
3 void foo (T x) { cout << x << " gt 4 bytes" << endl; }
4
5 template <typename T,
6         enable_if_t <(sizeof(T) <= 4)>*> = nullptr>
7 void foo (T x) { cout << x << " le 4 bytes" << endl; }
```

Здесь работоспособность основана на том, что параметры-указатели входят в анализ типов отдельной строчкой.

Фантастический пример использования `enable_if` для создания условного `explicit` был рассмотрен Стефаном Лававеем в его докладе на CppCon 2016.

## 4.10 Метапрограммирование

*To iterate is human, to recurse divine*

– L. Peter Deutsch

Шаблонное метапрограммирование было открыто в 1994-м году. Эрвин Анрух (Erwin Unruh) на заседании комитета по стандартизации сделал доклад, из которого следовало, что шаблоны могут быть использованы для вычисления на этапе компиляции и продемонстрировал генератор простых чисел, который на этапе компиляции выводил в виде сообщений об ошибках простые числа от 2 до заранее заданного настраиваемого предела. Но этот генератор довольно сложен. Кроме того, он представлял собой довольно неочевидный код.

### 4.10.1 Простая рекурсия и арифметика

Начать следует с очевидного: метапрограммирование это обобщение систематического SFINAE на целочисленные вычисления.

```

1 template<size_t N>
2 struct fact :
3     integral_constant<size_t,
4         N * fact<N - 1>{}> {};
5
6 template<> struct fact<0> :
7     integral_constant<size_t, 1> {};
```

Теперь в коде можно легко вычислить факториал пяти:

```
1 cout << fact<5>{} << endl;
```

Это вычисление не займёт ни секунды машинного времени: ответ будет известен уже на этапе компиляции.

Посмотреть как разворачиваются вычисления можно в специальной утилите metashell.

```
$ metashell
> #include <metashell/scalar.hpp>
> #include "meta-fact.hpp"
> #msh mdb factorial<6>::value
Metaprogram started
```

```
(mdb) ft
factorial<6>::value
+ factorial<6>
| + factorial<5>
| | + factorial<4>
| | | + factorial<3>
| | | | + factorial<2>
| | | | | + factorial<1>
| | | | | | + factorial<0>
| | | | | | | factorial<0>::(anonymous)
| | | | | | | | factorial<1>::(anonymous)
| | | | | | | | factorial<2>::(anonymous)
| | | | | | | | factorial<3>::(anonymous)
| | | | | | | | factorial<4>::(anonymous)
| | | | | | | | factorial<5>::(anonymous)
‘ factorial<6>
(mdb) quit
> #msh quit
```

Также легко организовать вычисление чисел Фибоначчи

```
1 template<size_t N> struct fibonacci :
2     integral_constant <size_t,
3             fibonacci<N-1>{} +
4             fibonacci<N-2>{}> {};
5
6 template<> struct fibonacci<1> :
7     integral_constant<size_t, 1> {};
8
9 template<> struct fibonacci<0> :
10    integral_constant<size_t,0> {};
```

Здесь двойная (древовидная) древовидная рекурсия уже может несколько смущать.

Не слишком ли много времени потратит на этот процесс компилятор?

#### 4.10.2 Две модели вычислений

Вернёмся к факториалу. Человек с опытом на С написал бы простую функцию, вычисляющую факториал числа, в итеративном стиле:

```

1 int
2 fact_0 (int x)
3 {
4     int i, res = 1;
5     for (i = 2; i <= x; ++i)
6         res *= i;
7
8     return res;
9 }
```

Здесь для того, чтобы вычислить факториал понадобились две ячейки памяти – изменяемые переменные `i` и `res`. Математик Аллан Тьюринг в 1936-м году опубликовал статью из которой следовало, что очень сложные вычислительные процессы можно произвести, имея в своем распоряжении:

- ветвления (скажем оператор `if` или тернарный оператор)
- переходы (хороший пример – любой цикл или простое `goto`)
- последовательное линейное исполнение инструкций
- достаточно большое количество изменяемой памяти

Факториал переписанный в более Тьюринг-стиле, но все ещё на С показывает некий минимализм выражительных средств:

```

1 int memory[3];
2
3 /* memory[0] is input and result */
4 fact_0:
5     memory[1] = 1;
6     memory[2] = 1;
7     label t, fin;
8 t:
9     memory[1] += 1;
10    memory[2] *= memory[1];
11    if (memory[1] > memory[0])
12        goto fin;
13    goto t;
14 fin:
15    memory[0] = memory[2];
```

Некоторое время спустя, Стивен Клини обнаружил, что очень сложные вычислительные процессы можно произвести, имея в своем распоряжении:

- ветвления (скажем оператор `if` или тернарный оператор)
- вызовы функций с передачей аргументов

В этом случае вообще не требуется изменяемых переменных.

И правда, факториал можно записать иначе:

```
1 int
2 fact_1 (int x)
3 {
4     if (x < 2)
5         return x;
6     else
7         return x * fact_1 (x - 1);
8 }
```

Такие функции были им названы **частично-рекурсивными** функциями и было доказано, что аппарат частично-рекурсивных функций эквивалентен по вычислительной мощности машинам Тьюринга и лямбда-выражениям

Порождаемый таким образом рекурсивный процесс вычислительно не слишком эффективен, поскольку выражение `x * fact_1 (x - 1)` не может быть вычислено до полного вычисления всех рекурсивных вызовов. Таким образом нет аккумулятора (вернее он неявно размазан по рекурсивной цепочке). Аккумулятор тоже можно сымитировать, изменив факториал:

```
1 int
2 fact_2_1 (int x, int idx, int product)
3 {
4     if (idx > x)
5         return product;
6     else
7         return fact_2_1 (x, idx + 1, product * idx);
8 }
9
10 int
```

```

11 fact_2 (int x)
12 {
13     return fact_2_1 (x, 1, 1);
14 }
```

Теперь в metashell картинка иная

```

Metaprogram started
(mdb) ft
factorial2<6>::value
+ factorial2<6>
| ` fact_rec<6, 2, 1>
|   ` fact_rec<6, 3, 2>
|     ` fact_rec<6, 4, 6>
|       ` fact_rec<6, 5, 24>
|         ` fact_rec<6, 6, 120>
` factorial2<6>
```

Интересно, что обычные программы на C++ больше похожи на подсахаренные машины Тьюринга (с поправкой на структурное программирование, взрослые циклы и вызовы функций). А вот метапрограммирование больше напоминает подход Клини.

Метапрограмма рекурсивного факториала несколько сложнее итеративной.

```

1 template <size_t n, size_t idx, size_t product>
2 struct fact_rec :
3     integral_constant<size_t,
4                     fact_rec <n, idx + 1, product * idx>{}> {};
5
6 template <size_t n, size_t product>
7 struct fact_rec <n, n, product> :
8     integral_constant<size_t,
9                     product * n> {};
10
11 template <size_t n>
12 struct factorial :
13     integral_constant<size_t, fact_rec <n, 1, 1>{}> {};
```

Но она же и более поучительна.

**Домашняя наработка:** сравните по скорости оба подхода на вашем компиляторе.

### 4.10.3 Ветвления и пример квадратного корня

Следующий пример куда более нетривиален: предлагается на этапе компиляции посчитать целочисленный квадратный корень.

Первый шаг для решения такой задачи: прикинуть код на обычном C++, но в функциональном стиле:

```

1 int
2 isqrt (int N, int lo = 1, int hi = N)
3 {
4     int mid = (lo + hi + 1) / 2;
5
6     if (lo == hi)
7         return lo;
8     else
9     {
10         if (N < mid * mid)
11             return isqrt (N, lo, mid - 1);
12         else
13             return isqrt (N, mid, hi);
14     }
15 }
```

Теперь этот код практически один в один может быть переложен на метапрограмму. Поскольку шаблонное инстанцирование даёт один уровень сопоставления с шаблоном, можно убрать самый верхний условный оператор.

Тогда завершение рекурсии будет выглядеть тривиально как `return lo` выше

```

1 template <int N, int S>
2 struct Sqrt <N, S, S, S> :
3     integral_constant<int, S> {};
```

Но что использовать для ветвления в общем шаге? На выручку приходит уже рассмотренный выше `conditional_t`.

```

1 template <int N, int L = 1, int H = N,
2             int mid = (L + H + 1) / 2>
3 struct Sqrt :
4     integral_constant<int,
5     conditional_t<(N < mid * mid),
```

```

6     Sqrt<N, L, mid - 1>,
7     Sqrt<N, mid, H>> {}> {};

```

Использование всё так же тривиально

```
1 cout << Sqrt<16>{} << endl;
```

Рассмотренные примеры показывают, что в метапрограммах можно делать ветвления и циклы (через рекурсию), хранить промежуточные данные и выполнять над ними основные арифметические операции. Всего этого достаточно для того, чтобы доказать Тьюринг-полноту шаблонов C++.

#### 4.10.4 Простые числа

Благодаря вложенным классам, можно устраивать некую модульность в вычислениях и даже писать нечто, похожее на “метаподпрограммы”. Большой пример с простыми числами сильно отличается от базового кода Анруха (его всегда можно найти в интернетах), но он несколько полезней, так как результаты печатаются не в виде ошибок, а доступны для использования.

```

// TODO: переписать систематичней

1 template <int i>
2 struct NthPrime
3 {
4     template <int p>
5     struct is_prime
6     {
7         template <int n>
8         struct n_divisors
9         {
10            template <int N, int M>
11            struct is_divisor
12            {
13                enum { val = is_divisor <N, M - 1>::val + ((N % M) ==
14                  0) };
15            };
16            template <int N>
17            struct is_divisor <N, 1>

```

```
18     {
19         enum { val = 0 };
20     };
21
22     enum { val = is_divisor <n, n - 1>::val };
23 };
24     enum { val = (n_divisors <p>::val == 0) };
25 };
26
27 template <int n, int m>
28 struct search_step
29 {
30     enum { val = search_step <n - (is_prime <m>::val), m + 1>::
31             val };
32 };
33
34 template <int m>
35 struct search_step <1, m>
36 {
37     enum { val = m - 1 };
38 };
39     enum { val = search_step <i, 3>::val };
40 };
```

Использование очевидно:

```
1 int main(int argc, char* argv[])
2 {
3     printf("Prime 6: %i\n", NthPrime<6>::val);
4     return 0;
5 }
```

**Домашняя наработка:** коды Грэя на шаблонах

## 4.11 Вычисления времени компиляции

*Once you're const, you are ensconced*

– Susan Powell

Вычисления времени компиляции до выхода стандарта C++11 велись исключительно с помощью особой шаблонной магии, рассмотренной выше. В новом стандарте появился способ сделать это по человечески. Новые ключевые слова `constexpr`, `constinit` и `constexpr` частично контролируют время выполнения выражения. Это позволяет создавать настоящие константы и константные выражения, заниматься метaproграммированием и многое другое. Здесь будут систематично рассмотрены все перечисленные возможности.

### 4.11.1 Ещё раз о константности

Уже рассматривавшийся выше модификатор `const` служит для того, чтобы объявить некие данные неизменяемыми. Но когда этим не изменяемым данным будет в первый раз присвоено их (в дальнейшем окончательное) значение? It depends.

```
1 const int MAXSIZE = numeric_limits<int>::max();
2 int arr[MAXSIZE]; /* not legal in C++ */
```

В этом примере значение максимального возможного целого будет присвоено `MAXSIZE` только в динамике. В результате на этапе компиляции, размер `arr` всё ещё неизвестен и строго соответствующие стандарту C++98 компиляторы обязаны выразить этим свое неудовольствие (такие массивы возможны в С и часто для C++ это включают как расширение).

Те же проблемы испытывают статические данные в классах и структурах:

```
1 struct S {
2     static const int sz;
3 };
4 const int page_sz = 4 * S::sz; // run-time
5 const int S::sz = 256;
```

Это законная запись. Инициализатор статической константы, как и было рассмотрено ранее, должен появиться вне класса. Далее `page_sz`

будет инициализирована верным значением, но потребует инициализации времени выполнения. Как ни странно, но совсем немного отличающийся код уже пройдет инициализацию на этапе компиляции:

```

1 struct S {
2     static const int sz = 256;
3 };
4 const int page_sz = 4 * S::sz; // compile-time
5 const int S::sz;
```

Это довольно грустно и требует от программиста помнить все тонкие правила константности, что, конечно, нереально. Именно этим и было мотивировано введение в стандарт C++11 ключевого слова, делающего известность на этапе компиляции явной.

Как вообще могла бы быть объявлена функция `numeric_limits<int>::max()` в C++98? Например так:

```

1 #define INT_MAX (2147483647)
2
3 template <>
4 struct numeric_limits<int> {
5     static inline int max () { return INT_MAX; }
6 };
```

Эта функция – прекрасный кандидат на формирование константного выражения. Она:

- не `void`, то есть возвращает какое-то значение
- состоит из одного `return` – то есть не заводит локальных переменных и не использует стек

Именно в таких двух условиях, функция в C++11 может быть сделана **константно-выраженной** для чего используется ключевое слово `constexpr`. Это ключевое слово отличает неизменность от неизменяемости.

```

1 template <>
2 struct numeric_limits<int> {
3     static constexpr int max () { return INT_MAX; }
4 };
```

Аналогично разворачивается пример со структурой: статическое константно-выраженное поле прекрасно работает.

```

1 struct S {
2     static constexpr int sz = 256;
3 };
4
5 constexpr int page_sz = 4 * S::sz;
6 constexpr int S::sz;
7 int arr[page_sz]; // ok

```

Но не каждый тип может быть так аннотирован. Уже для структуры `S` это не работает, `constexpr S = {}` будет ошибкой. Всё дело в том, что константные выражения могут образовывать только **литеральные** типы.

Литеральным считается тип у которого есть литералы, такие как `1`, `1.0` или `1ull`. Видно, что таких типов немного – в основном это примитивные типы такие как `int` и `float`. Вы можете расширять их перечень, но об этом далее.

Для константных выражений действуют статические асsertы времени компиляции.

```

1 constexpr int whole = 1;
2 constexpr double half = 0.5;
3 static_assert (half < whole, "Hmm...");
```

Самые большие сомнения вызывают константно-выраженные значения с плавающей точкой. Это вызвано тем, что результат операций над ними на этапе исполнения может не совпадать с результатом операций на этапе компиляции:

```

1 constexpr float ct = 1.0f / 3.0f;
2
3 void __attribute__((noinline))
4 magic_func (float x, float y)
5 {
6     fesetround (FE_DOWNWARD);
7     float rt = x / y;
8     if (ct != rt)
9         cerr << "Failure: " << ct << " != " << rt << endl;
10 }
11
12 magic_func (1.0, 3.0);
```

Здесь на экране будет показана просто шикарная ошибка: два визуально одинаковых числа оказываются неравны друг другу.

Константные выражения также имеют тёмные стороны для указателей и массивов. Например ниже приведена тривиальная, но не столь уж очевидная ошибка.

```
1 constexpr int arr[] = {2, 3, 5, 7, 11};
2 constexpr int *x = &arr[3]; // FAIL
```

Здесь справа от равенства стоит `const int*`, а слева стоит `int* const`. Поэтому, чтобы всё работало правильно, нужно использовать несколько безумно выглядящую запись.

```
1 constexpr const int *x = &arr[3]; // FAIL
```

Важно всегда отслеживать: константность **чего именно** в данном случае означает `constexpr` аннотацию.

### 4.11.2 Константно-выраженные функции

Более интересный пример: что если сделать константно-выраженной обычную арифметическую функцию?

```
1 constexpr int square(int x) { return x * x; }
2 /* ... */
3 constexpr int res = square(5);
```

Здесь `res` будет известен на этапе компиляции. Но при этом в таком виде:

```
1 // foo's argument unknown at compile time
2 int foo (int y) { return square(y); }
```

Функция `square` (даже объявленная `constexpr`) ведёт себя как самая обычная функция. Это позволяет иметь её одну, не заводя зоопарк `constexpr` и не-`constexpr` зверушек.

В стандарте C++14 ослаблены ограничения на константно-выраженные функции. Начиная с этого стандарта, в такой функции могут содержаться:

- Все объявления, кроме `static` и `thread_local` переменных, а также неинициализированных переменных

- Условные операции `if` и `switch`

Это делает написание таких функций гораздо более удобным. Например легальной становится следующая конструкция:

```

1 constexpr int
2 ipow (int x, int n)
3 {
4     int r = x;
5     while (--n > 0) r *= x;
6     return r;
7 }
```

Конечно это увеличивает работу для компилятора, но в сравнении с самой простой метапрограммой это выполняется куда быстрее и читается гораздо лучше.

Майерс в [7] приводит блестящий пример совмещения вывода типов шаблонами с `constexpr` функциями: определитель размера массива.

```

1 template <typename T, size_t N>
2 constexpr size_t arraySize (T(&) [N])
3 {
4     return N;
5 }
```

Работает это например так:

```

1 int keyVals[] = {2, 3, 5, 7, 11, 13};
2 int mapped[arraySize(keyvals)]; // same size
```

Здесь размер выводится из ссылки на массив аргумента.

### 4.11.3 Аннотация данных

Чтобы отличать обычные данные от известных на этапе компиляции, данные и даже члены классов тоже можно аннотировать `constexpr`

```

1 struct S
2 {
3     private:
4         static constexpr int sz; // constexpr variable
5     public:
```

```

6   constexpr int two(); //constexpr function
7 };
8
9 constexpr int S::sz = 256;
10 enum DataPacket
11 {
12     Small=S::two(), // error (call before def)
13     Big=1024
14 };
15
16 constexpr int S::two() { return sz*2; }
17 constexpr S s;
18
19 int arr[s.two()]; // ok (call after def)

```

Видно, что по существу они мало чем отличаются от обычных данных. Но компилятор имеет насчёт них точную гарантию что их значения при компиляции уже известны.

#### 4.11.4 Темные чудеса шаблонных переменных

В отличии от C++11, в C++14 разрешено делать шаблонными не только классы и функции, но и переменные:

```

1 template <typename T> T n = T(5);
2
3 int main() {
4     n<int> = 10;
5     cout << n<int> << " "; // 10
6     cout << n<double> << " "; // 5
7 }

```

Вместе с `constexpr` аннотацией переменных, это позволяет вводить гораздо более симпатичные определители типов, рассмотренные в (4.9.2).

```

1 template <typename T>
2 constexpr bool is_void_v = is_void<T>::value;

```

Записывать `is_void_v<T>` конечно гораздо проще и приятней, чем `is_void<T>::value` (общий смех).

Но тогда возникает вопрос. Неужели у нас могут быть `constexpr`

данные типов `int` или `float`, но не может быть `constexpr` данных пользовательских типов? Конечно могут.

#### 4.11.5 Аннотация конструкторов

Если у вас есть необходимость, чтобы ваш тип мог вести себя как compile-time константа, для него можно написать `constexpr`-конструктор:

```

1 struct Complex
2 {
3     constexpr Complex(double r, double i) :
4         re(r), im(i) { }
5
6     constexpr double real() const { return re; }
7     constexpr double imag() const { return im; }
8 private:
9     double re;
10    double im;
11 };
12
13 constexpr complex c(0.0, 1.0);

```

Тело такого конструктора в C++11 обязано быть пустым, вся инициализация производится в списке инициализации. В C++14 это правило несколько ослаблено.

Для такого пользовательского класса можно ввести даже арифметику:

```

1 struct Complex
2 {
3     constexpr Complex(double r = 0.0, double i = 0.0) :
4         re(r), im(i) { }
5
6     constexpr double real() const { return re; }
7     constexpr double imag() const { return im; }
8
9     constexpr Complex& operator+= (const Complex &rhs)
10    {
11        re += rhs.re;
12        im += rhs.im;
13        return *this;
14    }

```

```

14     }
15
16 private:
17     double re;
18     double im;
19 };
20
21 constexpr Complex operator+ (const Complex &lhs,
22                             const Complex &rhs)
23 {
24     Complex tmp = lhs;
25     tmp += rhs;
26     return tmp;
27 }
```

Но, конечно, всё это ООП времени компиляции является скорее изыском. Обычно такие абстракции в `constexpr` не нужны и гораздо большее применение имеют пользовательские литералы.

#### 4.11.6 Пользовательские литералы

Для определения пользовательского литерала, следует переопределить оператор кавычки.

```

1 constexpr Complex operator""_i( long double i )
2 {
3     return Complex (0.0, i);
4 }
```

Теперь будет работать следующий код:

```
1 constexpr Complex c = 0.0 + 1.0_i;
```

Таким образом будет (на этапе компиляции!) создана константа пользовательского типа.

СтроСтруп приводит даже более впечатляющий пример: <http://www.stroustrup.com/Software-for-infrastructure.pdf> (TODO: добавить в список литературы?)

```

1 template<int M, int K, int S> struct Unit {
2     enum { m=M, kg=K, s=S };
3 };
```

```

4
5 template<typename Unit>
6 struct Value {
7     double val;
8     explicit Value(double d) : val(d) {}
9 };

```

Далее ряд синонимов для тензорных индексов размерности:

```

1 using Meter = Unit<1,0,0>;
2 using Second = Unit<0,0,1>;
3 using Second2 = Unit<0,0,2>;
4 using Speed = Value<Unit<1,0,-1>>;
5 using Acceleration = Value<Unit<1,0,-2>>;

```

И пользовательские литералы для удобного обозначения констант.

```

1 constexpr Value<Meter> operator"" m(long double d)
2 {
3     return Value<Meter> (d);
4 }
5
6 constexpr Value<Second> operator"" s(long double d)
7 {
8     return Value<Second> (d);
9 }
10
11 constexpr Value<Second2> operator"" s2(long double d)
12 {
13     return Value<Second2> (d);
14 }

```

Написав соответствующие перегрузки операторов деления, далее можно добиться статической проверки типов в удобной форме:

```

1 Speed sp1 = 100_m/9.8_s; // ok
2 Speed sp2 = 100_m/9.8_s2; // error (m/s2 is acceleration)
3 Speed sp3 = 100/9.8_s; // error (100 has no unit)
4 Acceleration acc = sp1/0.5_s; // ok again

```

**Домашняя наработка:** добейтесь работы строчки `constexpr Kilogramm mass = 5_kg + 3_lb`, где lb это фунты, на этапе компиляции строящей преобразование фунтов в килограммы.

## 4.12 Домашняя наработка по шаблонам

### Контрольные вопросы

1. Чем отличается вывод типов шаблонами от вывода типов через `auto/decltype`?
2. Какие сложности вносит в разрешение перегрузки тот факт, что функция может быть шаблонной?
3. Как заставить компилятор инстанцировать шаблон функции для конкретного типа только один раз на все единицы трансляции?
4. Не нарушит ли шаблон функции в заголовочном файле ODR при инстанцировании везде куда включен заголовочный файл?
5. Чем отличается статический полиморфизм от динамического полиморфизма?
6. Может ли программа на C++ быть быстрее, чем такая же программа на C, реализующая тот же алгоритм?
7. Может ли шаблонный класс иметь несколько перегруженных конструкторов копирования?
8. Чем отличается полная специализация от частичной?
9. Как вы будете выбирать между специализацией и перегрузкой шаблонных функций?
10. Может ли быть полностью специализирован метод внутри определения частично специализированного класса?
11. В C++ нет частичной специализации функций. Можно ли сымитировать её средствами языка?
12. Возможен ли вывод типа при конструировании экземпляра шаблонного класса?
13. На каком этапе разрешаются зависимые имена в шаблонах классов?
14. Как внутри шаблонного класса сделать имя метода, не зависящего от шаблонных параметров, зависимым? Зачем это может быть нужно?

15. Можно ли сконструировать объект шаблонного типа при наличии конкурирующей специализации этого типа?
16. Можно ли в зависимости от шаблонного параметра изменить тип единственного поля в классе без частичной специализации этого класса?
17. Может ли шаблонная функция быть виртуальной?
18. Как расшифровывается СRTP, зачем нужна эта идиома?
19. Как в С++ происходит разрешение грамматической неоднозначности при использовании шаблонов?
20. Может ли вариабельный шаблон быть частично специализирован по части пачки параметров?
21. Перечислите все значения троеточия (ellipsis) в грамматике языка.
22. Можно ли раскрытием пачки параметров сформировать поля в классе?
23. В каких случаях пустая пачка параметров делает шаблонный класс не шаблонным?
24. Какие методы для нерекурсивного доступа ко всем параметрам пачки вам известны?
25. Напишите безопасный относительно типов scanf или аргументируйте невозможность этого.
26. Можно ли найти номер поля в std::tuple по значению?
27. Напишите вариабельный шаблон функции, которая для своей пачки параметров возвращает второй с конца аргумент
28. В каком случае вы предпочтете лямбда-функцию обычной? В каком наоборот?
29. Можно ли имея лямбда-функцию изменить что-то из захваченного ей по значению контекста и только потом вызвать её?
30. Можно ли захватить объект по правой ссылке (то есть по сути – передать его в лямбда-функцию)?
31. Можно ли запретить для отдельной лямбда-функции move-семантику?

32. В лекциях описаны проблемы с наивным подходом к перегрузке лямбда-функций и их решение. Но почему не работает наивный метод?
33. Меняет ли захваченный контекст тип лямбда-выражения?
34. Может ли повиснуть (dangle) правая ссылка на захваченный контекст?
35. Может ли лямбда-функция принимать переменное количество аргументов?
36. Может ли обобщенная лямбда-функция быть специализирована? Частично специализирована?
37. В чём обобщённые лямбды уступают шаблонам функций и в чём превосходят их?
38. Можно ли “перегрузить” лямбда функции, заставив вызов `foo(1)` против `foo(1.0)` работать иначе, где `foo` это лямбда и если да, то как?
39. Константно-выраженные лямбды в C++17: что это, зачем нужны, как работают.
40. Можно ли сымитировать захват по ссылке, располагая только захватом по значению с переименованием и если да, то как, а если нет, то чего не хватает?
41. Представим `static` переменную внутри лямбды. У объекта этой лямбды есть десять копий. Будет ли существовать девять статических переменных? Ответ обосновать.
42. Как написать рекурсивную лямбду и может ли она быть без захвата?
43. Иногда в коде перед лямбдами встречается значок плюса, например
  - 1   `+[] (int x) { return x*2}`Что это и зачем нужно?
44. Как бороться с возможным провисанием ссылок на захваченный контекст в крупных приложениях (где точно нереально просмотреть всё глазами)?

45. Что такое bind и какие его плюсы и минусы в сравнении с лямбдами?
46. Как расшифровывается SFINAE, зачем нужна эта идиома?
47. Говорят, что подстановка шаблонных параметров работает лениво. Что это означает?
48. Чем может быть параметризован шаблонный класс?
49. Что такое метапрограммирование на шаблонах? Напишите метапрограмму, вычисляющую числа Леонардо.
50. Напишите вычисление чисел Леонардо, используя integral constant.
51. Напишите вычисление чисел Леонардо как constexpr-функцию.
52. Представьте, что вы хотите иметь во время компиляции таблицу чисел Каталана. Как вы её себе организуете – метапрограммой на шаблонах, препроцессором или constexpr-функцией? Аргументируйте свой выбор.
53. Для объявления нового типа вы можете использовать using или typedef. В каком случае что вы будете использовать?
54. Может ли constexpr-конструктор быть в чисто абстрактном классе?
55. В лекциях приведен пример статического контроля размерности. Напишите упомянутую там перегрузку оператора деления.

## Задания

1. Реализовать поиск подпоследовательности в последовательности (алгоритм Кнута-Морриса-Прата) как обобщённую функцию
2. В разделе (2.2.1) был приведен совет (исходно принадлежащий Майерсу [5]) использовать где возможно `const` вместо `#define`. На это можно возразить, что, в случае отказа от препроцессора, простая проверка времени компиляции по значениям констант становится невозможной:

```
1 #define T1 2
2 #define T2 3
```

3

```
4  /* compile-time check here */
5  #if T1!=T2
6  #error "T1 and T2 are unequal"
7  #endif
```

Можно ли реализовать такую же проверку, определив T1 и T2 как статические константы, а не как макроопределения?

3. Напишите реверсирующую прозрачную оболочку: функцию, которая вызывает переданную ей функцию с переданными ей аргументами в обратном порядке. Страйтесь минимизировать оверхед.
4. Напишите рандомизирующую прозрачную оболочку: функцию, которая вызывает переданную ей функцию с переданными ей аргументами в случайном порядке (предположите что типы всех аргументов совместимые). Страйтесь минимизировать оверхед.
5. Напишите код, который применяет к вариабельной пачке из  $k^N$  параметров для  $k=1,2,\dots$  произвольную функцию N аргументов и возвращает кортеж из k результатов.
6. Напишите безопасный относительно типов scanf на вариабельных шаблонах
7. Напишите SFINAE-определитель, отвечающий на вопрос, есть ли в классе перегруженный оператор сложения
8. Напишите SFINAE-определитель, добавляющий (если это возможно) правую ссылку к типу или оставляющий тип неизменным. Скажем `int --> int&&`, но `int[10] --> int[10]`
9. Напишите три функции вида `int foo(T)`, параметризованные типом T, одна из которых инстанцируется только когда `sizeof(T) <= 2`, вторая если  $2 < sizeof(T) \leq 4$  и третья если `sizeof(T) > 4`. Все три должны существовать в коде одновременно, покрывая таким образом всё пространство вариантов.
10. Напишите на шаблонах метaproограмму, которая определяет старший значимый бит в числе на этапе компиляции.
11. Напишите на constexpr-функциях метапрограмму, которая определяет старший значимый бит в числе на этапе компиляции. Бонус за функцию, которая будет работать в C++11.

12. Спроектируйте простейшее compile-time отображение, параметризованное типами Key и Value.
13. Напишите `constexpr` функцию, берущую на вход `std::tuple` и извлекающую его вторую треть.
14. Напишите выбрасывающую прозрачную лямбду: лямбда-функцию, которая вызывает переданную ей функцию, передавая ей каждый второй, переданный в лямбду параметр.

# Глава 5

## Лабиринты стандартной библиотеки

*Most software today is very much like an Egyptian pyramid  
with millions of bricks piled on top of each other,  
with no structural integrity,  
but just done by brute force and thousands of slaves*

– Alan Kay

Идея что включать и что не включать в стандартную библиотеку языка программирования не так тривиальна, как может показаться.

Некоторые языки поставляются с библиотеками включающими всё, что может понадобиться когда бы то ни было, включая крайне высокоразвитые функции работы с XML, базами данных, сетевыми протоколами. Это философия таких языков как Python, Java, C#, которые идут “с батарейками внутри”.

Стандартная библиотека языка С (и некоторых других) придерживается, наоборот, философии минимальной поддержки, включая только те функции, которые могут пригодиться каждому разработчику (и некоторые функции, которые вызвали слезы няшного умиления у комитета стандартизации и оставлены, хотя не нужны, пожалуй, вообще никому, например strcspn).

Язык С++ занимает промежуточную позицию. По своей задумке его библиотека придерживается второго подхода, но по факту от стандарта к стандарту асимптотически сходится к первому.

Стандартная библиотека языка C++ имеет долгую и интересную историю.

Идея стандартизировать библиотеку в языке С (из которой позже выросла стандартная библиотека C++) возникла не сразу. Первая стандартизация библиотеки языка С это стандарт ANSI С 1989-го и последовавший за ним ISO/IEC 9899-1990 [20]. Эта библиотека известна под названиями libc или crt (от C run-time support) и в общем она довольно стабильна уже много лет и активно используется программистами. Но в этой библиотеке никогда не было ничего, похожего на контейнеры. Каждый раз, когда программисту на языке С нужен двусвязный список... ну что же, он пишет себе двусвязный список. В лучшем случае он где-нибудь находит готовый и молится, что найденное работает.

В самом начале пути языка C++, ещё на до-стандартном этапе, стало ясно, что возможности нового языка позволяют включить **обобщённые** контейнеры в том или ином виде. Первые шаги были сделаны когда в языке появились стандартные строки и потоки ввода-вывода.

Где-то к 1992 году, Алекс Степанов из Adobe, переписал с языка Ada на C++ свой проект, который он назвал STL – стандартная библиотека шаблонов.

Этот проект был весьма странным для того времени: если язык C++ как таковой делался со строгим упором на ООП, Степанов разносил данные от методов их обработки, делал ортогональные итераторы и алгоритмы и в общем использовал скорее идеи функционального, чем объектно-ориентированного программирования. Тем не менее, результат получился превосходным и очень быстро, ещё до стандартизации в 98-м году библиотека STL стала индустриальным стандартом де-факто.

Одновременно с развитием контейнеров, пришло понимание роли исключений и важности написания кода с учётом исключений. К 2002-му году основные вопросы безопасности исключений в библиотеке были решены.

С тех пор развитие стандартной библиотеки не стоит на месте и в 2011 а потом и в 2017 она была несколько раз существенно расширена и переработана. Была добавлена даже поддержка работы с файловыми системами, обсуждается добавление средств работы с популярными сетевыми протоколами, были серьёзно пересмотрены аллокаторы.

В силу исторических особенностей своего формирования, стандартная библиотека языка C++ получала свои возможности слой за слоем. Сейчас это настоящий лабиринт, в котором, зная путь, можно сделать

удивительные вещи удивительно просто.

В силу опыта чтения лекций, я считаю нужным начать не с контейнеров и не с итераторов, а с самой простой вещи, которая может быть на свете. Со строк. Тем более, они уже упоминались (см. 2.7.2)

## 5.1 Строки

В самом общем виде, строка это массив символов с известной длиной. На самом деле, любой массив символов, известный на этапе компиляции, параметризован своей длиной как частью типа.

Например `decltype("Hello, world!")` выведет `const char (&)[14]`

**Вопрос к студентам:** а почему вывелась ссылка?

Можно обратить внимание, что “Hello, world!” содержит тринадцать символов. Четырнадцатым является завершающий ноль, который неявно прислан в конец каждого строкового литерала.

Строка с завершающим нулём называется C-строкой (C-string). Вся унаследованная от языка С через хедер `<cstring>` часть стандартной библиотеки С работает именно с C-строками.

Конечно, при работе со строками размер которых неизвестен на этапе компиляции, нужна некоторая унификация типов. Она достигается для C-строк через приведение их к указателю на первый элемент. Благодаря этому строки могут жить как в глобальной памяти, так и в куче или на стеке.

```
1 const char *cinv = "Hello, world";
2 char cmut[] = "Hello, world";
3 char *cheap = malloc /* some size */;
```

Здесь `cinv` это указатель на первый элемент живущего в глобальной памяти строкового литерала (который поэтому является неизменным). При его инициализации копирования не происходит. С другой стороны, `cmut` это строка на стеке или в изменяемой глобальной памяти. При её инициализации делается копирование строкового литерала в ячейки памяти и сама строка далее является изменяемой. Также изменяема и строка `cheap`, но она уже явно выделена на куче под нужный размер.

Нет никаких проблем перебрасывать данные между разными типами памяти, скажем `strcpy(cheap, cinv)` работает без проблем.

Две наиболее частые проблемы это утечка ресурсов при бездумном присвоении указателя и попытка присвоить указатель массиву (к счастью последнее это просто ошибка компиляции).

```
1 cheap = cinv; // LEAK
2 cmut = cheap; // FAIL
```

Разумеется никто не запрещает взять `char *pcmut = cmut` и далее `pcmut = cheap` разумеется будет работать. Все технические проблемы здесь это обычные проблемы с массивами и указателями, они уже рассматривались ранее (см. 2.4).

### 5.1.1 Проблемы безопасности в С-строках

Когда при мне говорят, что в языке С строки представляют собой небольшую проблему в безопасности, я всегда выслушиваю это с тем же видом, с каким выслушал бы информацию о том, что тропический ливень, град и гроза представляют небольшую опасность намокнуть.

```
1 void apply_world (char *sout, const char *sin) {  
2     strcpy(sout, sin);  
3     strcat(sout, ", world!");  
4 }
```

**Вопрос к студентам:** какие вы видите здесь проблемы с безопасностью? Ответ будет сразу ниже, не читайте его прежде чем сами не насчитаете хотя бы четыре пункта.

Коротко говоря, тут плохо всё. В первую очередь мы не можем быть уверены, что `sout` указывает вообще на что-то разумное. Во-вторых даже если это так, не факт, что там достаточно памяти чтобы скопировать туда `sin` и даже если достаточно, то в третьих не факт, что `sin` указывал на нечто разумное и ограниченное по размеру. И даже, если указывал, то, в четвёртых, никто не знает сколько времени займёт копирование. И даже если немного, то, в пятых, неясно и нет возможности проверить хватит ли в `sout` места на то, чтобы дописать туда ещё и кусок литерала.

Слишком много беспокойства о двух строчках кода, не так ли?

Пытаясь как-то учесть эти проблемы, люди сделали специальные функции с ограничением на размер буфера.

```
1 char* strncpy (char *dst, const char *src, size_t n);  
2 char* strncat (char *dst, const char *src, size_t n);  
3 int strncmp (const char *s1, const char *s2, size_t n);
```

Но это слабое утешение.

Во-первых есть функции (скажем `strlen`) для которых так не сделать в принципе. А во вторых это всё равно не решает проблему подсовывания вместо осмысленного буфера чего угодно на вход функции.

Настоящая причина проблем – в том, что для С строки длина не является инвариантом. Чтобы сохранять инварианты таких объектов как строки, необходимо закрытое состояние, недоступное к модификации, т.е. необходима инкапсуляция. Что естественным образом приводит к идее: написать класс строки

**Вопрос к студентам:** вы хотели бы написать собственный класс строки с нуля?

Ответ на этот вопрос я предлагаю отложить до того момента, как мы познакомимся с тем как устроен класс строки в стандартной библиотеке.

То есть `std::string`.

### 5.1.2 Принципиальное устройство `string`

Чтобы эффективно оперировать со строкой, нужно понимать её принципиальное устройство. Картинка (рис. 5.1) весьма условна и неполна, но она отражает тот взгляд, который может иметь достаточно прагматичный программист на строки в своей программе.

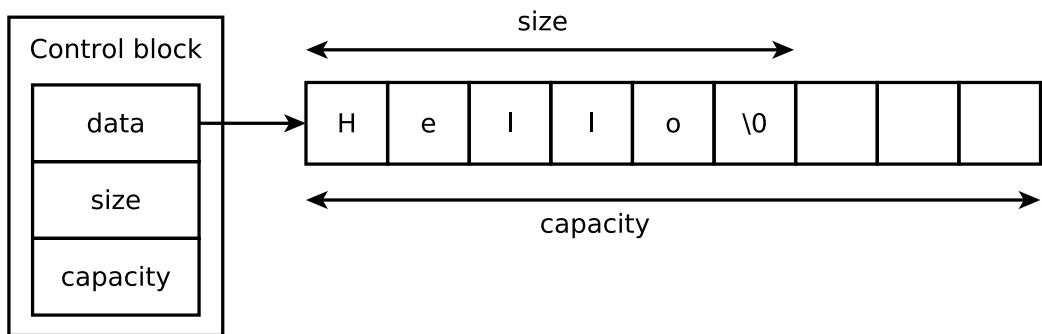


Рис. 5.1: Принципиальное устройство строки

Одна из странных вещей на этой картинке: здесь очевиден двойной контроль длины строки. Стока завершается нулём и, одновременно, её размер хранится как поле класса. Это сделано для эффективной работы метода `c_str`, в частности, чтобы вообще иметь возможность сделать его `const` методом.

Это в общем обычный класс, его методы всегда можно подсмотреть в документации. Для строк переопределены основные операторы (такие как сложение и сравнение на равенство), а также неизбежные копирование присваивание и перемещение.

В качестве чуть более интересного примера можно рассмотреть поиск в строке. Для этого есть сразу три метода:

- `find` – возвращает позицию подстроки в строке
- `find_first_of` – возвращает позицию первого совпадения строки с одним из символов подстроки
- `find_first_not_of` – возвращает позицию первого несовпадения с одним из символов подстроки

Метод `find` возвращает позицию как число, то есть номер символа в строке, являющемся началом найденной (в случае удачи) подстроки. При неудачном поиске возвращается специальное значение `string::npos`. Важно помнить, что тип, в котором происходит поиск это обычно беззнаковый тип, так что `npos` это чаще всего очень большое беззнаковое число. Поэтому важно не промахнуться с типом для его хранения. При поиске в строках следует всегда использовать `string::size_type` иначе вы нарываетесь.

```

1 string s = "Hello";
2 szt notfound = s.find("bye");
3 assert (notfound == string::npos);
4 szt ellp = s.find("ell");
5 assert (ellp != string::npos);

```

Иллюстрация работы этого кода приведена на (рис. 5.2).

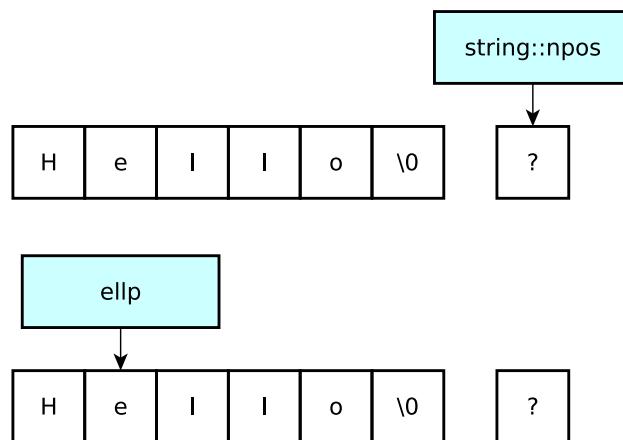


Рис. 5.2: Поиск в строках

Возможен также поиск с конкретной позиции

```

1 szt hpos = s.find("H", ellp);
2 assert (hpos == string::npos);

```

Разумеется начиная с позиции `ellp` никакой буквы “H” в строке нет, поэтому она не найдена.

Простые задачи поиска просты и в С стиле. Например, приведённый выше код может быть переписан.

```

1 const char s[] = "Hello";
2 char *notfound = strstr("bye", s);
3 assert (!notfound);
4 char *ellp = strstr("ell", s);
5 assert (ellp);
6 char *hpos = strstr(H, ellp);
7 assert(!hpos);

```

Выгода начинает быть видна когда задачи становятся немного (немного!) менее тривиальными.

Каноническая задача – заменить в строке все подстроки данного вида. Скажем в строке `"Hello, $username, how are you doing, $username?"` заменить все вхождения `$username` на `Eric, the Bloody Axe`.

**Задача у доски:** написать это на C++ в стиле С без `std::string`.

Задача непроста и требует некоторого размышления. Между тем на современном C++ она решается очень просто (даже не используя алгоритмов стандартной библиотеки, просто с помощью встроенных средств типа `string`)

```

1 void replace_all (string& str, const string& from,
                    const string& to)
2 {
3     size_t ncur = 0;
4     while ((ncur = str.find (from, ncur)) != string::npos) {
5         str.replace (ncur, from.length(), to);
6         ncur += to.length();
7     }
8 }
9 }

```

В реальности эта функция будет чуть сложнее из-за контроля входных данных (например случая `from.empty()`), но идея именно такая. Использовать её просто и приятно.

```

1 string str = "Hello, $username, how are you doing, $username?";

```

```
2 string from = "$username";
3 string to = "Eric, the Bloody Axe";
4 replace_all (str, from, to);
5 cout << str << endl;
```

Конечно, вариант с C-строками можно написать так, что он выиграет у варианта с C++ строками представленного выше, но только на экстремально длинных строках, которые, в общем, редкость. Но да, в любом случае это надо учитывать. Автору приходилось убирать perf issues из кода, который занимался копированием сотен тысяч небольших строк через оператор присваивания класса `string`. Об оптимизациях для небольших строк речь пойдёт позже.

### 5.1.3 Проблемы владения содержимым

Иногда в программе требуется константная статическая строка: нечто, что выделяется один раз и более не изменяется. В приступе эйфории от удобства строк, начинающий программист вполне может написать следующий код.

```
1 static const string kName = "FOO";
2 // ....
3 int foo(const string &arg);
4 // ....
5 foo(kName);
```

**Вопрос к студентам:** чем он плох?

Вопрос в общем прост. Сложно догадаться задать его себе. Страна живёт в динамической памяти (т.н. “куче”). Строковый литерал `"FOO"` будет скопирован в свежевыделенное место кучи. После этого компилятор навсегда потеряет возможность статически доказывать нечто о его дальнейшей судьбе. Это называется `heap indirection`.

Допустим, проблема осознана и в качестве шага к её исправлению делается следующий: строка заменяется на указатель.

```
1 static const char *kName = "FOO";
2 // ....
3 int foo(const string &arg);
4 // ....
5 foo(kName);
```

Это то, что китайские товарищи называют “и было плохо а стало хуже некуда”. Теперь при вызове функции мы должны создать новый экземпляр временной строки, которая будет связана с константной ссылкой в формальном параметре.

**Вопрос к студентам:** как вы относитесь к идеи существенно улучшить ситуацию, заменив в сигнатуре функции ссылку на строку на указатель на С-строку?

Итак, кажется, достойного решения этой проблемы нет. Вернее не было. К настоящему времени проблема была осознана и в стандарт C++17 был добавлен вариант невладеющей строки `string_view`. Его принципиальное устройство показано на (рис. 5.3).

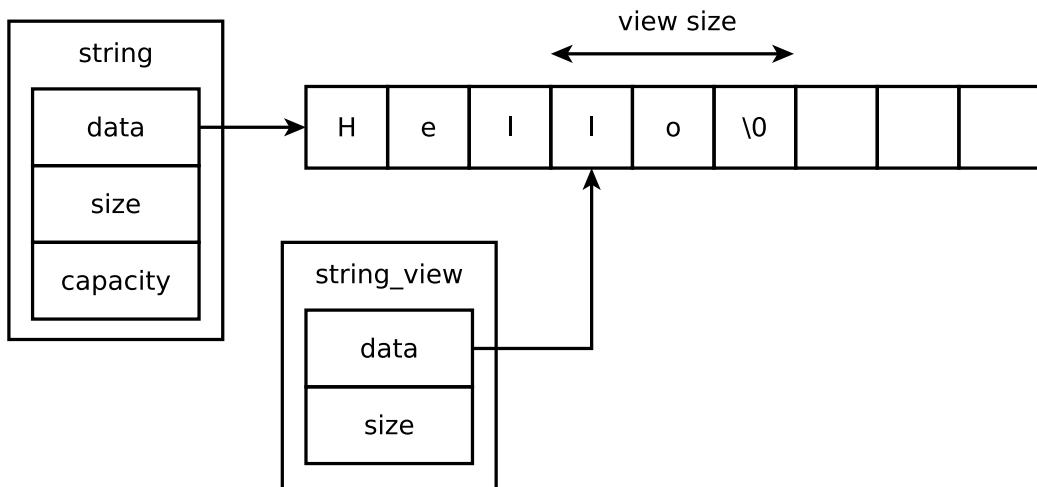


Рис. 5.3: Принципиальное устройство `string view`

С его помощью рассмотренная проблема решается просто, очень просто.

```

1 static const string_view kName = "FOO";
2 // ....
3 int foo(const string_view &arg);
4 // ....
5 foo(kName);
  
```

В первой строчке нет heap indirection, поскольку `string_view` не владеет своим содержимым и значит не должен иметь его локальную копию. По той же причине в последней строчке нет создания временного объекта, как и нет проблем с типизацией и безопасностью, присущих строкам в стиле С.

Базовые методы для `string_view` это:

- `remove_prefix` и `remove_suffix` – позволяют сдвинуть окно view слева и справа
- `copy` – копирует подстроку в буфер (крайне опасный метод)
- `substr` – возвращает view на подстроку
- `compare` – около шести перегрузок этого метода делают различные сравнения
- `find` – поиск по view
- `data` – данные под view в сыром виде

Чтобы почувствовать как на самом деле выглядит работа с отображениями строк, полезно что-нибудь попробовать.

```
1 string str = " trim me ";
2 string_view vtrim = str;
3 auto trimfst = vtrim.find_first_not_of(" ");
4 vtrim.remove_prefix(min(trimfst, vtrim.size()));
5 auto trimlst = vtrim.find_last_not_of(" ");
6 vtrim.remove_suffix(vtrim.size() - min(trimlst, vtrim.size()));
```

Отображения строк облегчают проблему владения, но не решают её. Следующий код мог бы написать человек с опытом Python.

```
1 std::string a = ssl ? "https" : "http";
2 a += "://";
3 a += path;
4 a += "/";
5 a += query;
```

Этот код вполне легален на C++ но давайте проследим как это будет работать.

Сначала будет создана строка `a` и туда будет скопирован один из литералов. Далее будет создана временная строка из `a + "://"` и в ней будет скопировано содержимое. Потом то же самое повторится ещё трижды. После этого результат будет скопирован в строку `a`.

Это слишком много копирований. Хуже того: это слишком много ре-аллокаций памяти (а динамическую память дешево использовать, но довольно дорого выделять и освобождать). Гораздо лучше мы бы отнеслись к коду следующего вида:

```
1 std::string a =  
2     combine(ssl ? "https" : "http", "://", path, "/", query);
```

Но что это за функция `combine`? В стандарте ничего такого нет. Самый простой вариант: сделать её строковым потоком. Но можно написать и настоящий комбинатор с использованием вариабельных шаблонов. При разговоре про потоки можно будет вернуться к тому как могла бы выглядеть такая функция (см. 5.5.8).

Ещё хуже тот факт, что часто нет возможности дать **гарантию**, что строка не владеет содержимым. Небольшая строка, допустим, уходит в какую-нибудь функцию. Она там может измениться, так что не может быть отображением, но чаще всего на практике она там не изменяется и таким образом владение было оплачено зря.

```
1 void foo (string s);  
2 string s1 = "Hello";  
3 foo (s1);  
4 string s2 = s1;  
5 foo (s2);
```

Разумеется, такие случаи были замечены и активно обсуждались ещё с ранних 90-х. Одним из перспективных путей решения этого вопроса тогда казались строки с совместным использованием.

#### 5.1.4 Строки с совместным использованием

В конце прошлого раздела была поставлена проблема совместного использования памяти строками. Основная идея здесь получила называнием “copy on write” или сокращённо COW. Идея заключается в том, что строки совместно владеют одной и той же памятью пока они одинаковые и при этом достаточно умны, чтобы понять когда нужно перейти к единоличному владению если одной из них требуется изменение (см. рис. 5.4)

Идея совместно используемых строк очевидно имеет свои плюсы и минусы.

К плюсам относятся:

- Экономия памяти
- Дешёвое копирование (просто инкремент счётчика ссылок)

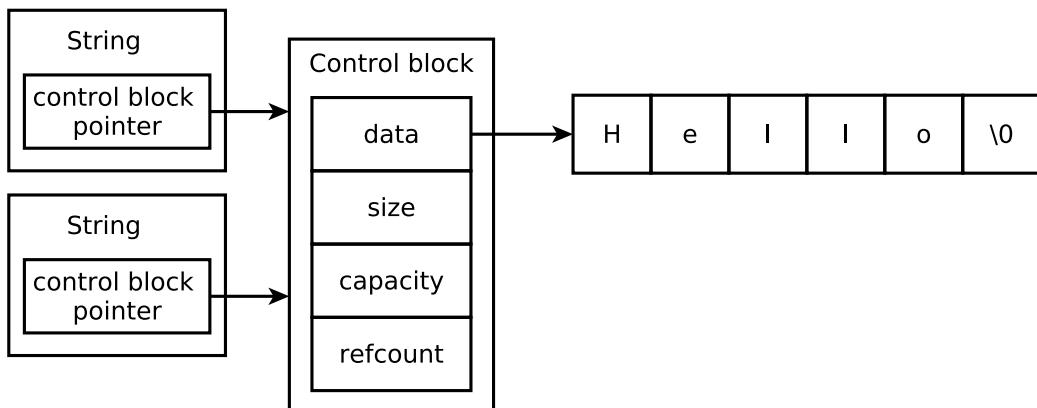


Рис. 5.4: Идея copy on write строк

- Меньше аллокаций и удалений в куче => прирост производительности

К минусам:

- Лишний уровень косвенности
- Вирусное проникновение копирования во все модифицирующие операции
- Проблемы thread safety (Multithread COW disease)

Эта идея была столь популярна, что была использована даже в GCC. Стока в GCC до версии 5 выглядела как показано на (рис. 5.5).

Споры насчёт таких строк не утихали до тех пор, пока не был найден решающий аргумент: аргумент от **инвалидации** указателей.

Дело в том, что при операциях над строкой некие указатели могут становиться недействительными.

```

1 string s = "Hello";
2 const char *p = &s[3];
3 s += "world"; // p became invalid

```

В целом тут нет никаких проблем. Проблемы (проблемы!) начинаются когда совместно используемые строки (и только они) начинают инвалидировать указатели при операциях, в целом выглядящих крайне безобидно.

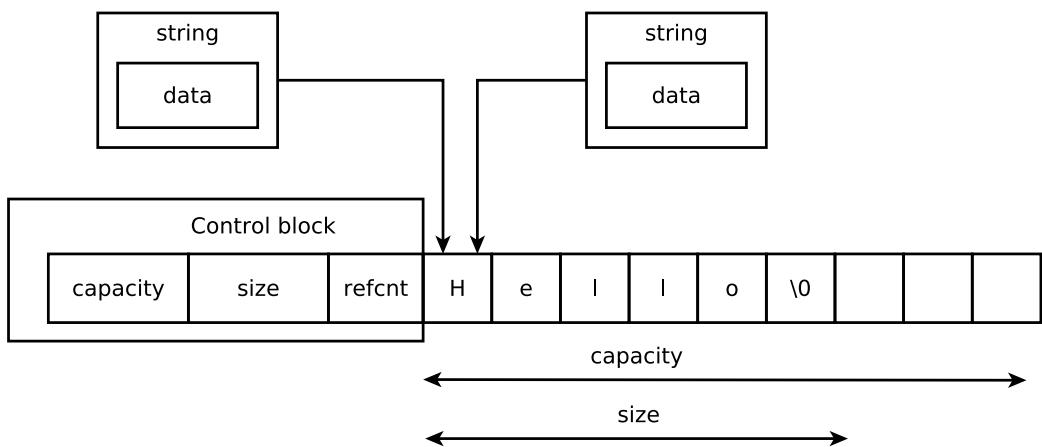


Рис. 5.5: Реализация copy on write строк в libstdc++ v5

```

1 string s = "Hello";
2 const char *p = &s[3];
3 s[0] = 'h'; // p valid for non-COW
4 // p invalid for COW

```

В 2011 году официально было запрещено (TODO: stand link?) инвалидировать указатели при выполнении `operator[]`, что исключает COW-реализации `std::string`.

В итоге идиома COW находится в упадке: вне стандарта COW is almost dead.

### 5.1.5 Оптимизации для небольших строк

Ещё одной идеей как оптимизировать работу со строками являются оптимизации для небольших строк (SSO). Действительно, большинство строк в реальных программах невелики. Самый распространённый паттерн это буквально пара слов текста на каком-нибудь человеческом языке. Идея в следующем: если строка занимает меньше динамической памяти, чем поддерживающие структуры, может быть стоит и хранить её прямо на стеке в контрольном блоке? Идея проиллюстрирована на (рис. 5.6).

Переводя это в код, можно написать нечто вроде:

```

1 class string {
2     size_type m_size;

```

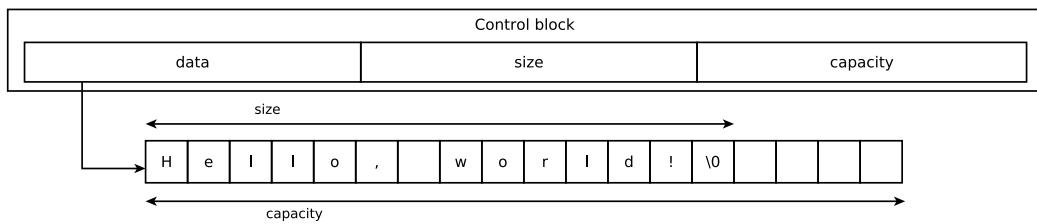


Рис. 5.6: Идея для small string optimizations: настоящий масштаб для схемы строки

```

3   union {
4     class {
5       char *m_data;
6       size_type m_capacity;
7     } m_large;
8     char m_small[sizeof(m_large)];
9   };
10 // ....
11 };

```

В SSO можно увидеть и плюсы (они очевидны – меньше реаллокаций и меньше места в куче) и минусы. Минусы это в первую очередь:

- Усложняется копирование и (что важнее) перемещение
- Добавляется время на выбор `m_small` / `m_large` при каждом доступе (в том числе чтении) с проверкой размера

Вторая проблема серьёзней. В GCC она решается так, как показано на (рис. 5.7)

Это решение позволяет избежать потерь времени при доступе, но уменьшает размер самой строки.

### 5.1.6 Обобщения строк

Пока что весь разговор вёлся в предположении, что строка это набор символов `char`. Но вообще-то это не так. В некоторых кодировках символ занимает не один байт, а два (как в UCS-2) или четыре (как в UCS-4). Можно себе представить экзотические кодировки, где символ вообще будет иметь пользовательский тип или тип с плавающей точкой.

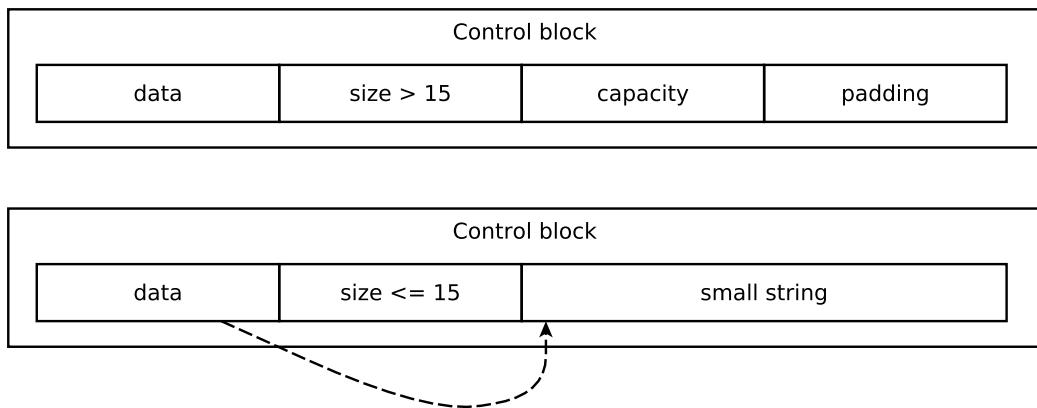


Рис. 5.7: SSO строка в libgcc версии выше пятой

К счастью решение этой проблемы известно и это шаблоны. В первом приближении шаблон класса строки может выглядеть как-то так.

```

1 template <typename CharT> class basic_string {
2     CharT *data;
3     size_t size;
4     union {
5         size_t capacity;
6         enum {ALIGN = 32 - sizeof(data) - sizeof(size); }
7         enum {SZ = (sizeof(capacity) + ALIGN - 1) / ALIGN;};
8         enum {NUNITS = SZ / sizeof(CharT); }
9         CharT small_str[NUNITS];
10    } sso;
11 public:
12     // all 89 methods goes here
13 };

```

Для удобства четыре его инстанциации могут быть определены через конкретные подстановки.

```

1 typedef basic_string<char> string;
2 typedef basic_string<u16char_t> u16string;
3 typedef basic_string<u32char_t> u32string;
4 typedef basic_string<wchar_t> wstring;

```

Но это не отменяет необходимости реализовать все 89 методов, среди которых могут быть нетривиальные. Самые простые вопросы: как сравнивать символы, что такое завершающий символ, как копировать символы – это вопросы, ответы на которые сложно получить, не зная ничего

о символе.

По традиции вся эта информация выносится в класс характеристик (traits).

```
1 template <typename CharT> class char_traits;
```

Методы в этом классе: `assign`, `eq`, `lt`, `eof` и прочие, позволяют тонко настроить характеристики для конкретного типа символов.

Во втором приближении (с учётом характеристик), шаблон строки может быть переписан.

```
1 template <typename CharT,
2         typename Traits = std::char_traits<CharT>>
3 class basic_string {
4 // all the same with traits
```

Но прежде чем можно будет умыть руки, нужно предусмотреть ещё одну деталь: выделение памяти на символ. Это явно не может быть свойством самого символа, потому что на одни и те же символы можно выделять память разными способами и с разными стратегиями. Но это не может быть также свойством строки по тем же причинам. Поэтому это обычно отделяется в специальный класс аллокатора.

```
1 template <typename CharT,
2         typename Traits = std::char_traits<CharT>
3         typename Allocator = std::allocator<CharT>>
4 class basic_string {
```

Об аллокаторах речь пойдёт далее (см. 5.10).

**Домашняя наработка:** пожалуй теперь вы знаете достаточно, чтобы построить свой велосипед строки. Сделайте это.

## 5.2 Исключения

*The most effective debugging tool is still careful thought,  
coupled with judiciously placed print statements.*

– Brian W. Kernighan, 1979

В предыдущем разделе ничего не было сказано про обработку ошибок при работе со строками и это не случайно. Стока это класс, а обработка ошибок при работе с классами требует нелокальных по природе механизмов, в частности исключений. Кое-что об исключениях уже было сказано ранее (см. 2.8), но подробное изложение последует только в этом разделе, поскольку в стандартной библиотеке исключения использованы практически везде и настала пора понять зачем это нужно и как это работает.

Самым удивительным выводом этого раздела будет тот факт, что использование механизма исключений серьёзно влияет на проектирование кода: и на открытый интерфейс классов и на их внутреннее устройство. Хорошее понимание исключений открывает двери к пониманию многих проектных решений в стандартных контейнерах и в библиотеке вообще. Кроме того очень часто при поддержке унаследованного кода обнаруживаются скрытые ошибки, допущенные его менее опытными авторами именно в безопасности относительно исключений. Тем больше поводов внимательно разобраться в этом сложном и важном языковом механизме.

### 5.2.1 Обработка ошибок в конструкторах

Классическая обработка ошибок кодами возврата в стиле C ломается в C++ на специальных функциях, у которых непонятен контекст возврата. Самый простой пример это конструктор.

```
1 template <typename T> class MyBuffer {
2     T *arr_ = nullptr;
3     size_t size_, used_;
4 public:
5     MyBuffer (size_t sz): size_(sz), used_(0) {
6         arr_ = (T*) malloc (sizeof(T) * sz);
7         // here case of arr_ == nullptr shall be processed
8     }
```

Как обработать ситуацию исчерпания памяти?

Прежде чем отвечать на этот вопрос, можно подумать над другим вопросом: а собственно чем нам грозит эта ситуация, если её не обработать? Угроза на самом деле есть и она очень неприятная.

```
1 MyBuffer b(100);
```

После этой строчки объект `b` может оказаться в **несогласованном состоянии** (inconsistent state). У него будут выставлены поля `v.arr_ = 0` т.к. память кончилась, но, при этом `v.size_ = 100`, т.к. конструктор никак не обработал ошибку.

Несогласованное состояние это бич объектно-ориентированных контекстов. Хуже всего то, что объект в несогласованном состоянии никак не отличается от нормального объекта: это даже не UB, такое состояние объекта вполне легально. Но инвариант класса нарушен и последствия от этого могут проявится через тысячи строк кода и тысячи часов исполнения.

Итак, обработка необходима, при этом вернуть код ошибки нет возможности. Можно поступить так, как поступают потоки ввода-вывода (см. 5.5) и завести в классе особое поле `state`, устанавливая там какой-нибудь `failbit` или `badbit`. Но, конечно, это очень плохая тактика, потому что все такие механизмы разделяют ещё одну проблему с кодами ошибки: они **тихие**.

Объект, оказавшись в несогласованном состоянии должен информировать об этом создающий его контекст настолько громко, чтобы у контекста не было возможности проигнорировать эту информацию. В случае же кодов ошибок и, тем более, состояний потоков, более 90% кода, который видел автор, просто игнорируют проверку.

На помощь приходят исключения.

```
1 MyBuffer (size_t sz): size_(sz), used_(0) {
2     arr_ = (T*) malloc (sizeof(T) * sz);
3     if (!arr_) throw std::bad_alloc();
4 }
```

Здесь `std::bad_alloc` это стандартный класс исключения. Этот код может быть даже упрощён, учитывая, что обычное поведение оператора `new` это как раз сообщить исключением об исчерпании памяти

```
1 MyBuffer (size_t sz): size_(sz), used_(0) {
2     arr_ = new T[sz]; // throws bad_alloc
```

```
3 }
```

Человек, впервые оценивший всю мощь исключений, часто начинает лепить их по поводу и без повода. Важно отличать ошибки от исключительных ситуаций и правильно обрабатывать то и другое.

Исключительные ситуации характеризуются двумя важнейшими признаками:

- Состояние программы должно быть восстановимо
- Исключительная ситуация не может быть обработана на том уровне, на котором возникла

Легко видеть, что в приведённом примере оба условия выполнены. Разумеется если нарушено первое условие, то лучшее, что можно сделать это завершить программу. Если же ошибка может быть обработана на том же уровне, она должна быть обработана, механизм исключений в этом случае является слишком дорогим и избыточным.

Всё дело в том, что порождение любого исключения инициирует размотку стека.

### 5.2.2 Размотка стека

Что выведет на экран исполнение следующего кода?

```
1 struct UnwShow
2 {
3     UnwShow () { printf ("ctor\n"); }
4     ~UnwShow () { printf ("dtor\n"); }
5 };
6
7 void
8 foo (int n)
9 {
10     UnwShow s;
11     if (n == 0) throw 1;
12     foo (n - 1);
13 }
14
15 /* ... */
16 foo (3);
```

Казалось бы здесь программа забирается глубоко в рекурсию и оттуда бросает исключение раньше чем какой-либо из созданных объектов выйдет из поля зрения. Тем не менее, на экране видны не только четыре конструктора, но и четыре деструктора! Что произошло? Произошедшее называется **stack unwinding** и по русски обозначается как **размотка** или **раскрутка** стека.

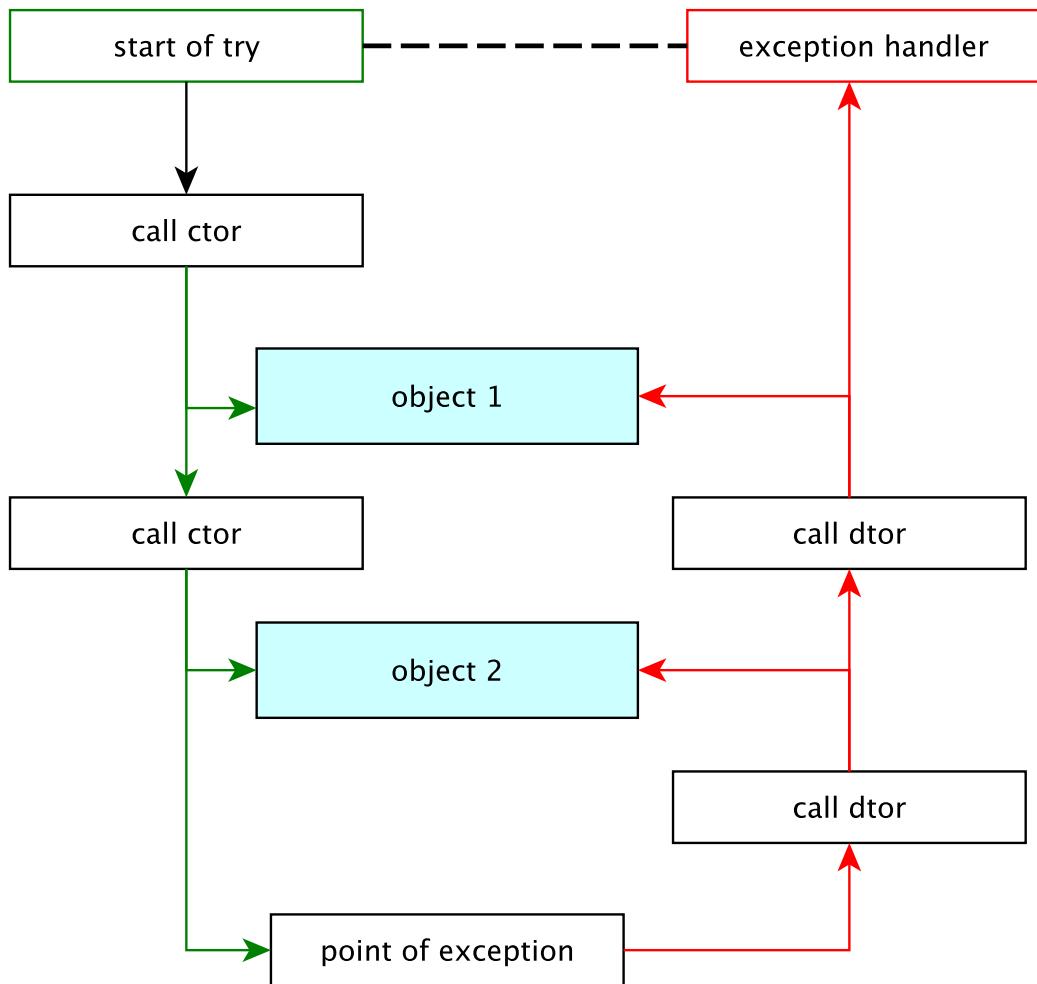


Рис. 5.8: Stack Unwinding

Процесс показан на (рис. 5.8). Довольно сложно показать на статичном рисунке динамический процесс раскрутки, но идея кажется ясной – все объекты, созданные от места входа в обработку исключений (пока что можно считать что просто от начала программы) до обработчика удаляются со стека корректно – то есть с вызовом деструкторов.

Раскрутка стека продолжается до тех пор, пока не будет найден первый подходящий `catch`, которому и передаётся управление.

Ловля происходит по точному типу

```
1 try { throw 1; } catch(long l) {} // NOT CATCH
```

Или по ссылке на точный тип

```
1 try { throw 1; } catch(const int &ci) {} // CATCH
```

Или по указателю на точный тип

```
1 try { throw new int(1); } catch(int *pi) {} // CATCH
```

Или по ссылке или указателю на базовый класс

```
1 try { throw Derived(); } catch(Base &b) {} // CATCH
```

Catch-блоки пробуются в порядке перечисления

```
1 try { throw 1; }
2 catch(long l) {} // NOT CATCH
3 catch(const int &ci) {} // CATCH
```

Пойманную переменную можно менять или удалять

```
1 try { throw new Derived(); } catch(Base *b) { delete b; } // ok
```

Пойманное исключение можно перевыбросить

```
1 try { throw Derived(); } catch(Base &b) { throw; } // ok
```

Ниже следует очень плохой пример, так не надо делать никогда:

```
1 class MathErr /* ... */;
2 class Overflow : public MathErr /* ... */;
3
4 void foo()
5 {
6     try {
7         /* ... dangerous code here ... */
8     }
9     catch (MathErr) {
10        /* ... dealing with other guys ... */
11    }
12    catch (Overflow) {
```

```

13     /* ... dealing with overflow ... */
14 }
15 }
```

В приведённом примере исключения перехватываются по значению. Это очень плохо и ведёт к проблеме срезки, рассмотренной в (3.7.1). Хорошо написанный код перехватывает исключения по константной ссылке или по указателю (если это конечно не исключения POD-типов, которые программист также имеет право возбуждать, их можно перехватывать по значению).

Важно понимать, что обработчики проверяются по порядку перечисления. Поэтому ставить обработчик `MathErr` выше чем `Overflow`, означает сделать последний чуть более чем бесполезным.

Итак правильный порядок: от частных к общим и ловить строго по косвенности

```

1 catch (Overflow& o) { /* ... dealing with overflow ... */ }
2 catch (MathErr& e) { /* ... dealing with others ... */ }
```

**Вопрос к студентам:** как вы относитесь к ловле исключений по указателю?

### 5.2.3 Стандартные классы исключений

Хороший тон это генерировать потомка `std::exception` или одного из его потомков в иерархии стандартной библиотеки.

На (рис. 5.9) слева зелеными обозначены те классы, которые могут быть выброшены в процессе работы со стандартной библиотекой. Справа – два основных класса, от которых вам предлагается наследовать свои классы-обработчики.

```

1 class MathErr : public runtime_error {/* ... */};
2 class Overflow : public MathErr {/* ... */};
3
4 void foo()
5 {
6     try {
7         /* ... dangerous code here ... */
8     }
9     catch (const Overflow &) {
```

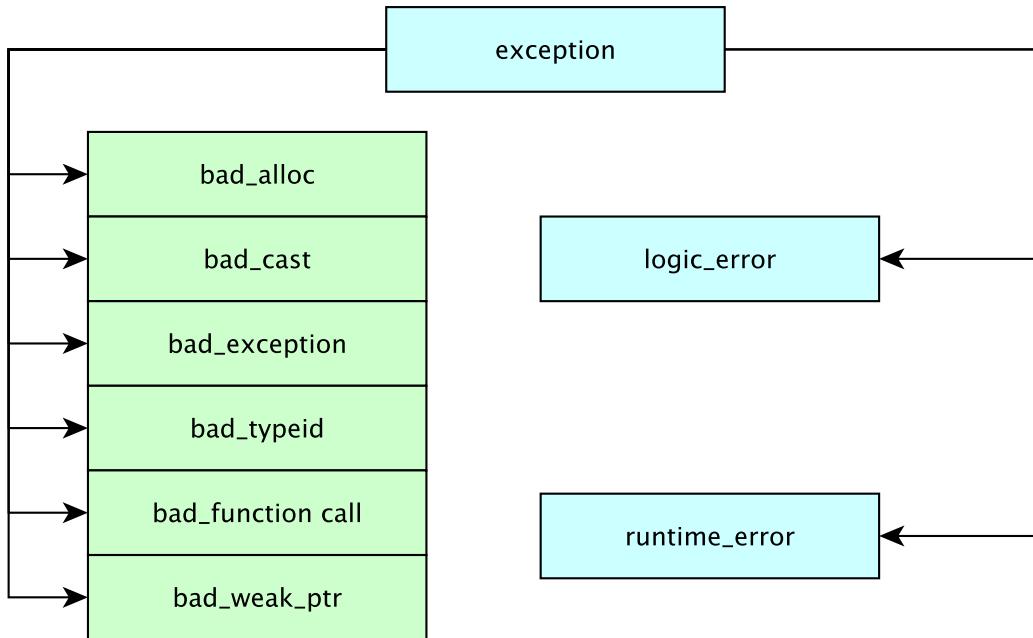


Рис. 5.9: Иерархия стандартных исключений

```

10     /* ... dealing with overflow ... */
11 }
12 catch (const MathErr &) {
13     /* ... dealing with other guys ... */
14 }
15 }
```

Такой вариант кода гораздо более правилен и плюс есть доступ к многочисленным полезным методам стандартных классов, которые они предоставляют. Самый простой пример это метод `what()` у класса `std::exception`, который возвращает строку с которой исключение было возбуждено

```

1 throw runtime_error("Parameter value not allowed in context");
2
3 // a lot of code here
4
5 catch(exception const& e) { cout << e.what() << endl; }
```

При использовании стандартных классов рекомендуется быть осторожным с множественным наследованием

```
1 struct my_exc1 : exception {
```

```
2     char const* what() const;
3 };
4
5 struct my_exc2 : exception {
6     har const* what() const;
7 };
8
9 struct your_exc3 : my_exc1, my_exc2 {};
```

Это конструкция опасна, потому что в итоге произойдёт нечто странное.

```
1 throw your_exc3();
2
3 // a lot of code here
4
5 catch(exception const& e) { cout << e.what() << endl; }
6 catch(...) { cerr << "whoops!\n"; }
```

**Вопрос к студентам:** вы понимаете почему будет whoops?

Как уже было сказано (см. 2.8.1) у программиста есть соблазнительная возможность перехватить в своём обработчике не конкретные классы исключений, а вообще все исключения, используя для этого специальный синтаксис с троеточием. Здесь эта возможность была использована по делу, но вообще ей надо пользоваться с большой осторожностью, потому что в общем случае такой перехват может нарушить нейтральность функции относительно чужих исключений.

О нейтральности стоит поговорить отдельно.

### 5.2.4 Нейтральность

Важным свойством исключительной ситуации является то, что она в принципе может быть обработана на каком-то уровне. Но чтобы быть обработанным, исключению нужно туда долететь.



Рис. 5.10: Нарушение нейтральности

На (рис. 5.10) показан случай, когда функция (обозначенная жёлтым) сообщает о проблеме, другая функция, обозначенная зелёным, может её обработать, но функция посередине просто ликвидирует нормальное движение объекта исключения вниз по стеку, перехватывая и свои и чужие исключения.

Такие функции называются нарушающими нейтральность.

Для сохранения нейтральности относительно исключений даже при необходимости поймать все исключения, пойманное исключение нужно дальше перевыбросить.

```
1 int *critical = new int[10000]();
2 try {
3     // a lot of dangerous code
4 }
5 catch (...) {
6     delete [] critical;
```

```
7     throw;  
8 }
```

В этом фрагменте кода рассмотрен, пожалуй, единственный рациональный случай когда действительно нужно поймать все исключения и очистить критичный ресурс если что-то пошло не так.

**Вопрос к студентам:** а можно здесь вс-таки что-нибудь придумать чтобы обойтись без catch-all?

### 5.2.5 Гарантии безопасности

*Exception handling isn't hard. Error handling is hard*

– David Abrahams

Исключения, являясь частным случаем нелокальной управляющей конструкции, добавляют строчки в неявный контракт на каждый метод и добавляют неожиданные дуги возможного выхода в каждом месте вызова небезопасной с точки зрения генерации исключений функции. Поэтому при проектировании кода, серьёзно использующего исключения, программист несёт дополнительную интеллектуальную нагрузку. Он должен следить за тем, что называется безопасностью кода относительно исключений.

Например для рассмотренного выше класса `MyBuffer` очевидный конструктор копирования скорее всего будет в некотором смысле плох.

```

1 template <typename T> class MyVector {
2     T *arr_ = nullptr;
3     size_t size_, used_;
4 public:
5     MyVector (const MyVector &rhs) {
6         arr_ = new T[rhs.size_]; // we will leak this memory
7         size_ = rhs.size_; used_ = rhs.used_;
8         for (size_t i = 0; i != rhs.size_; ++i)
9             arr_[i] = rhs.arr_[i]; // if exception is thrown here
10    }

```

Этот пример был обнаружен Каргиллом в 1994 году [29] и подробно рассмотрен Абрамсом в 1998 [30] и Саттером в 2000 и 2002 годах [13], [14].

Код, в котором при исключении могут утечь ресурсы, оказаться в несогласованном состоянии объекты и прочее, называется небезопасным относительно исключений.

Существует три гарантии безопасности, которые может предоставлять код:

- **Базовая гарантия** заключается в том, что сбой при выполнении операции может изменить состояние программы, но не вызывает утечек и оставляет все объекты в согласованном (но не обязательно предсказуемом) состоянии, т.е. пригодными к дальнейшему ис-

пользованию.

- **Строгая гарантия** обеспечивает транзакционную семантику: при сбое операции гарантируется неизменность состояния программы относительно задействованных в операции объектов.
- **Гарантия бессбойности** означает, что функция не генерирует исключений.

Например для решения проблемы Каргилла, можно ввести функцию осуществляющую безопасное копирование.

```

1 template <typename T>
2 T *safe_copy (const T* src, size_t srcsize) {
3     T *dest = new T[srcsize];
4     try {
5         for (size_t idx = 0; idx != srcsize, ++idx)
6             dest[idx] = src[idx];
7     }
8     catch (...) {
9         delete [] dest;
10        throw;
11    }
12    return dest;
13 }
```

Теперь с использованием этой функции конструктор окажется безопасен в пределах базовой гарантии.

```

1 MyBuffer (const MyBuffer &rhs) {
2     arr_ = safe_copy (rhs.arr_, rhs.size_);
3     size_ = rhs.size_; used_ = rhs.used_;
4 }
```

**Вопрос к студентам:** а может быть в пределах строгой гарантии?

Обычно в случае перемещения можно предоставить самую строгую гарантию: при перемещении `MyBuffer` речь идёт о побитовом копировании указателя, тут просто ничего не может случиться. Про спецификатор `noexcept`, являющийся языковым средством указания гарантии бессбойности уже было сказано (см. 2.8.2).

```

1 MyBuffer (MyBuffer &&rhs) noexcept :
2     arr_(rhs.arr_), size_(rhs.size_), used_(rhs.used_)
```

```

3  {
4      rhs.arr_ = nullptr;
5      rhs.size_ = 0; rhs.used_ = 0;
6  }
7
8 MyBuffer& operator= (MyBuffer &&rhs) noexcept
9 {
10     swap (arr_, rhs.arr_);
11     swap (size_, rhs.size_);
12     swap (used_, rhs.used_);
13 }
```

Интересная задача здесь это написать оператор присваивания. Менее опытный программист может начать делать это в наивном стиле.

```

1 MyBuffer& operator= (const MyBuffer &rhs) {
2     if (this == &rhs) return *this;
3     delete [] arr_;
4     arr_ = safe_copy(rhs.arr_, rhs.size_);
5     size_ = rhs.size_; used_ = rhs.used_;
6     return *this;
7 }
```

Этот подход довольно хрупкий и опасный. Например в приведённом выше примере кода допущена ошибка.

**Вопрос к студентам:** найдите ошибку в операторе присваивания.

Распространённый трюк здесь это реализовать присваивание в терминах уже реализованного копирования и перемещения. Это даёт транзакционную семантику и строгую гарантию для присваивания.

```

1 MyBuffer& operator= (const MyBuffer &rhs) {
2     MyBuffer tmp (rhs); // copy-ctor
3     std::swap (*this, tmp); // move-ctor, move-assign
4     return *this;
5 }
```

Таким образом конструктор копирования, оператор присваивания и перемещение дают примеры всех трёх гарантий. Но на самом деле проектирование с использованием исключений это не только о гарантиях безопасности. Подлинное влияние исключений на классы оказывается куда глубже.

К тому же, хочется обратить внимание, что весь код выше был построен на функции `safe_copy`, которая содержит внутри себя перехват всех исключений, что крайне некрасиво. Чуть позже будет предложен метод лучшего проектирования таких классов, а пока следует просто задаться вопросом: влияют ли исключения на проектирование.

### 5.2.6 Влияние исключений на проектирование

Для ответа на этот вопрос, проще всего рассмотреть следующую задачу: пусть хочется для класса `MyBuffer` написать метод `pop` который возвращает последний элемент сохранённый в памяти и удаляет его из буфера.

```
1 T pop () {
2     assert(used_ > 0);
3     T result = arr_[used_ - 1];
4     used_ -= 1;
5     return result;
6 }
```

Кажется, что этот метод несложен и всё хорошо. Но что произойдёт при попытке его использовать?

```
1 MyBuffer<SomeType> v;
2 // a lot of code here
3 SomeType s = v.pop();
```

Допустим здесь произошло исключение при копировании в `s`. Тогда получается удивительная ситуация: объект благополучно “отбыл” и был удалён из контейнера но не прибыл по месту назначения. Чтобы избежать таких ситуаций, принято делать отдельно метод, который извлекает элемент из контейнера и отдельно метод который удаляет его оттуда.

```
1 T top () {
2     return arr_[used_ - 1];
3 }
4
5 void pop () {
6     used_ -= 1;
7 }
```

Итак, исключения влияют на проектирование.

В этом случае почему бы не спроектировать `MyBuffer` сразу таким образом, чтобы не испытывать проблем с безопасностью исключений?

В этом нам в частности помогут особые формы оператора `new`, рассмотренные ранее (см. 3.4.2) в частности размещающий `new`.

Два полезных хелпера это создание объекта в сырой памяти

```
1 template <typename T, typename ... Ts> void
2 construct (T *p, Ts&& ... values) {
3     new (p) T (forward<Ts>(values)...);
4 }
```

И разрушение такого объекта без освобождения памяти

```
1 template <class T> void
2 destroy(T* p) noexcept {
3     p->~T();
4 }
5
6 template <typename FwdIter>
7 void destroy(FwdIter first, FwdIter last) noexcept {
8     while (first != last) {
9         destroy(&*first);
10        ++first;
11    }
12 }
```

Разумеется, такая функция `destroy` может вызвать обоснованную критику: она очевидно не защищена от ситуации когда деструктор уничтожаемого объекта выбрасывает исключение. Программист мог бы попробовать переписать её в защищённом стиле:

```
1 template <typename FwdIter>
2 void destroy(FwdIter first, FwdIter last) noexcept {
3     while (first != last) {
4         try
5         {
6             destroy( &*first );
7         }
8         catch (...)
9         {
10            /* what to do here? */
11        }
12 }
```

```

12     ++first;
13 }
14 }
```

Основная проблема – что делать в обработчике? Есть три идеи и все три очень плохи.

1. Можно снова генерировать в обработчике перехваченное исключение. Тогда функция удовлетворяет гарантии нейтральности... и, пожалуй, всё. При этом `destroy` не сможет сообщить о количестве корректно уничтоженных объектов и не уничтоженные останутся не уничтоженными, что вызовет утечку ресурсов. К тому же придётся убрать `noexcept`.
2. Можно, перехватывая исключение, генерировать некое другое исключение. Тогда функция не удовлетворяет гарантии нейтральности и всё равно может вызвать утечку
3. Можно, перехватывая исключения, как-то гасить их (может быть с некоторой обработкой). Но тогда функция не удовлетворяет гарантии нейтральности и, с точки зрения вызывающей функции, скрывает ошибки

Итак, если деструктор `T` может генерировать исключения, всё свидётся к написанию в лучшем случае небезопасного кода. Этой проблеме также подвержены невинно выглядящие `new[]` и `delete[]`, о которых шла речь в (2.7.4).

```

1 T *arr = new T[10];
2 delete [] arr;
```

**Вопрос к студентам:** как будет вести себя этот код, если предположить, что деструктор `T` может генерировать исключения?

Вывод из этого довольно прост: никогда не пишите типы, деструкторы которых, могут сгенерировать исключения.

Теперь всё готово, чтобы спроектировать безопасный относительно исключений буфер.

### 5.2.7 Двухуровневый контейнер

Основная идея двухуровневого контейнера, подобного разобранному выше `MyBuffer` в разделении управления сырой памятью от интерфейса.

Поскольку подобным же образом устроены многие стандартные контейнеры, в частности `std::vector` имеет смысл назвать такой контейнер `MyVector`, чтобы намекнуть на решения в интерфейсе и реализации.

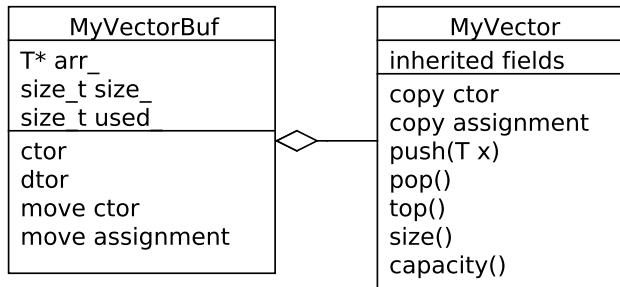


Рис. 5.11: Диаграмма классов двухуровневого буфера

Основная идея показана на (рис. 5.11). Все поля хранятся в `MyVectorBuf`

```

1 template <typename T>
2 struct MyVectorBuf
3 {
4     MyVectorBuf(const MyVectorBuf&) = delete;
5     MyVectorBuf& operator= (const MyVectorBuf&) = delete;
6     protected:
7         T *arr_;
8         size_t size_, used_;
9     // .... etc ....

```

Очень важно, что базовый класс не является копируемым. Это позволяет избежать всех проблем, связанных с копированием по умолчанию.

Конструктор и деструктор просто выделяют и освобождают сырую память.

```

1 MyVectorBuf(size_t sz = 0) :
2     arr_(sz == 0) ? nullptr :
3         static_cast<T*>(::operator new(sizeof(T) * sz)),
4     size_(sz), used_(0) {}
5
6 ~MyVectorBuf() noexcept {
7     destroy(arr_, arr_ + used_);
8     ::operator delete(arr_);
9 }

```

На самом деле специально помечать деструктор как `noexcept` не нужно, начиная с C++11 все деструкторы помечены так автоматически.

Функция `destroy`, использованная здесь была описана чуть выше. Перемещение и перемещающий конструктор тривиальны.

Базовый класс `MyVectorBuf` связан с наследником `MyVector` закрытым наследованием как показано ниже.

```

1 template <typename T> struct MyVector :
2                     private MyVectorBuf<T>
3 {
4     using MyVectorBuf<T>::used_;
5     using MyVectorBuf<T>::size_;
6     using MyVectorBuf<T>::arr_;
7 // .... etc ....

```

Разумеется, закрытое наследование это и есть композиция. Собственно можно обойтись без него и сделать композицию явной, но ценой некоторого неудобства дальнейших записей (`buf.used_` вместо `used_` и т.п.).

Теперь копирующий конструктор выглядит изящно и не требует `catch-all`

```

1 MyVector (const MyVector &rhs) : MyVectorBuf<T>(rhs.used_) {
2     while (used_ < rhs.used_) {
3         construct (arr_ + used_, rhs.arr_[used_]);
4         used_ += 1;
5     }
6 }

```

Функция `construct` это просто обёртка над размещающим `new` и была показана ранее. Здесь следует остановиться подробней, потому что именно в этот момент становится ясна красота замысла.

Что будет, если внутри `construct` возникнет исключение? Ничего страшного, поскольку теперь память будет корректно освобождена деструктором базового подобъекта.

Очень красиво также выглядит метод `MyVector::push` в такой архитектуре

```

1 void push(T t) {
2     if (used_ == size_) {
3         MyVector tmp (size_*2 + 1);
4         while (tmp.size() < used_)

```

```

5     tmp.push(arr_[tmp.size()]);
6     tmp.push(t);
7 // ----- Kalb line -----
8     swap(*this, tmp);
9     return;
10 }
11 construct(arr_ + used_, t);
12 used_ += 1;
13 }
```

Комментарием в коде помечена линия, которая называется “линией Калба” (Kalb line). Это полезная идиома при проектировании безопасного кода. Считается, что выше линии Калба код может генерировать исключения, но не влияет на состояние класса, а ниже неё – влияет на состояние, но не генерирует исключений.

Итак, хорошее проектирование кода позволяет сделать его безопасным относительно исключений. Примерно таким образом будет устроена большая часть контейнеров стандартной библиотеки.

Конечно реализовать всё это можно только если есть хотя бы некоторые функции, такие как `swap`, которые отвечают гарантии бессбойности и не кидают исключений никогда.

Что если пойти дальше и распространить бессбойность на прочие части кода?

### 5.2.8 Жизнь без исключений

Кое что о спецификаторе `noexcept` уже было рассказано (см. 2.8.2) но уже в расках этого раздела мы несколько раз были вынуждены обратиться к гарантии бессбойности, в основном в коде деструкторов или коде, который используется деструкторами (таком как функция `destroy`).

На самом деле, многие интересные возможности использовать `noexcept` были пропущены (по определённым причинам).

Можно ещё разок посмотреть на метод `push` и на то как он копирует объекты в новую память:

```

1 MyVector tmp (size_*2 + 1);
2 while (tmp.size() < used_)
3     tmp.push(arr_[tmp.size()]);
```

```
4 tmp.push(t);
5 // ----- Kalb line -----
6 swap(*this, tmp);
7 return;
```

Это подозрительный код, потому что непонятно зачем **копировать** нечто, что потом сразу же будет уничтожено? Почему не **перемещать** это? С перемещением всё выглядит чуть лучше

```
1 MyVector tmp (size_*2 + 1);
2 while (tmp.size() < used_)
3     tmp.emplace(move(arr_[tmp.size()]));
4 tmp.emplace(move(t));
5 // ----- Kalb line -----
6 swap(*this, tmp);
7 return;
```

Но некоторым оверхедом выглядит тот факт, что перемещение делается дважды. Первый раз при заполнении `tmp`, а второй раз в перемещающем присваивании, когда будет отрабатывать `swap`.

Можно улучшить время ещё вдвое если просто выделить на `arr_` больше памяти и заполнить сразу её. В этом случае, конечно, непонятно, что делать с исключениями, если они вылетят при перемещающем присваивании.

Но что если они не вылетят?

Кажется здесь есть возможность действовать двумя способами: более консервативным, если перемещающее присваивание порождает исключения и менее консервативным, если оно их не порождает. Для того, чтобы отличать такие случаи, в язык был введен оператор `noexcept(X)` который возвращает `true` или `false` в зависимости от того, может выражение `X` бросить исключение или не может. А сама аннотация `noexcept` для метода сделана условной.

В итоге если есть странная функция, которую непонятно помечать `noexcept` или нет, например такая:

```
1 template <class T>
2 T copy(T const& original) // noexcept?
3 {
4     return original;
5 }
```

То, во-первых можно использовать условный `noexcept` обратив внимание, что для фундаментальных типов тут явно не будет сбоев

```

1 template <class T>
2 T copy(T const& original) noexcept(is_fundamental<T>::value)
3 {
4     return original;
5 }
```

А во-вторых (и это лучший вариант), можно просто использовать оператор внутри условной директивы и сказать, что какой бы ни был тип, если его копирование не порождает исключений, то и эта функция тоже не должна.

```

1 template <class T>
2 T copy(T const& original) noexcept(noexcept(T(original)))
3 {
4     return original;
5 }
```

Оператор `noexcept` оценивает каждую функцию, задействованную в выражении, но не вычисляет выражение

```

1 struct ThrowingCtor { ThrowingCtor(){} };
2 void foo (ThrowingCtor) noexcept;
3 void foo (int) noexcept;
4 assert (noexcept (foo(1)) == true);
5 assert (noexcept (foo(ThrowingCtor{})) == false);
```

Разумеется он также возвращает `false` для constant expressions

**Домашняя наработка:** переписать метод `push` в двух SFINAE перегрузках, используя условный `noexcept`.

Жизнь без исключений не означает полного отказа от обработки ошибок. Для ошибок, которые могут быть обработаны на том же уровне прекрасно работают старые добрые коды возврата, которые в C++11 были даже улучшены с помощью нового заголовочного файла `<system_error>`.

Новые стандартные типы `error_code` и `error_category` и список стандартных категорий в пространстве имён `std::errc` дают возможность унифицировать семантические группы кодов ошибок (как это делается например при ловле исключений по базовому классу).

```
1 error_code ec { MY_OUTOF_MEM, errc::not_enough_memory };
```

Здесь `errc` это категория ошибки, а `error_code` может быть платформенно зависимым. Теперь сравнение на равенство сравнивает код со своей группой

```
1 if (ec == errc::not_enough_memory) {  
2     // process not enough memory  
3 }
```

Использование перегрузки по кодам ошибки позволяет иметь две функции, одна из которых бросает исключения а вторая сообщает код ошибки в одном классе.

```
1 void push (T new_elem);  
2 void push (T new_elem, error_code &ec) noexcept;
```

На самом деле заголовочный файл `<system_error>` гораздо больше и интересней. Интересующиеся студенты могут изучить его на досуге и доложить результаты, например на семинаре.

### 5.3 Умные и слишком умные указатели

*Manual lifetime management really can seem akin  
to constructing a mnemonic memory circuit  
using stone knives and bear skins*

– Scott Meyers

Идиома RAII уже рассматривалась в (3.2.1), и там же была сделана наивная попытка завернуть в класс для управления ресурсами такой классический разделяемый ресурс как дисковый файл. Конечно, идея обобщить RAII для любого  $T$ , используя шаблоны – приходит на ум почти сразу после знакомства с шаблонами.

```

1 template <typename U>
2 int foo (int x) {
3     U *ptr = new U(x);
4     // .... some code ....
5     if (some condition) {
6         delete ptr;
7         return -1;
8     }
9     // ....
10    delete ptr;
11    return 0;
12 }
```

Этот мотивирующий пример плох, кажется, вообще всем: здесь ручное управление ресурсами с дублированием кода, здесь небезопасность относительно исключений и всё такое.

Хотелось бы завернуть это в некую симпатичную RAII обёртку

```

1 template <typename U>
2 int foo (int x) {
3     RAIIPtr<U> ptr(new U(x));
4     // .... some code ....
5     if (some condition) {
6         return -1;
7     }
8     // ....
9     return 0;
10 }
```

Паттерн обертки, осуществляющей управление временем жизни объекта умнее, чем это делается через обычный указатель, называется “smart pointer” или умный указатель. Таким образом это не какой-то класс, это общая концепция для целого семейства от крайне простых до крайне сложных и интересных классов.

### 5.3.1 Первая попытка – не слишком умный указатель

Первая попытка написать не слишком сложную обертку для управления ресурсами, может выглядеть как-то так:

```

1 template <typename T> class RAIIPtr {
2     T *ptr_;
3 public:
4     RAIIPtr(T *ptr = nullptr) : ptr_(ptr) {}
5     ~RAIIPtr() { delete ptr_; }
6     T operator*() const { return *ptr_; }
7     T* operator->() const { return ptr_; }
8 };

```

Перегрузка двух интересных операторов – `operator*` (dereferencing, разыменование) и `operator->` (indirection, косвенный доступ) уже обсуждалась ранее (см. 3.3).

Тем не менее, в коде выше есть проблема и она становится очевидной, если посмотреть на него через “волшебные очки” (про этот удивительно полезный девайс, см. 3.2.2).

```

1 template <typename T> class RAIIPtr {
2     T *ptr_;
3 public:
4     RAIIPtr(T *ptr = nullptr) : ptr_(ptr) {}
5     ~RAIIPtr() { delete ptr_; }
6     RAIIPtr(const RAIIPtr&) = default;
7     RAIIPtr& operator= (const RAIIPtr&) = default;
8     T operator*() const { return *ptr_; }
9     T* operator->() const { return ptr_; }
10 };

```

Теперь должно быть очевидно, что обычное копирование или создание по образцу такого указателя приведет к ошибке двойного удаления

(см. 3.2.1, где эта проблема изложена подробно).

Конечно, если содержимое может быть скопировано, его можно скопировать. Это подход неинтрузивных контейнеров, таких как `vector` или `string`. Но в общем случае содержимое не может быть так просто скопировано. Что тут можно сделать?

Самый простой способ – запретить копирование и присваивание.

```

1 template <typename T> class SRAIIIPtr {
2     T *ptr_;
3 public:
4     SRAIIIPtr(T *ptr = nullptr) : ptr_(ptr) {}
5     ~SRAIIIPtr() { delete ptr_; }
6     SRAIIIPtr(const SRAIIIPtr&) = delete;
7     SRAIIIPtr& operator=(const SRAIIIPtr&) = delete;
8     T operator*() const { return *ptr_; }
9     T* operator->() const { return ptr_; }
10 };

```

Эта стратегия называется “scoped pointer” и именно так устроен и работает класс `boost::scoped_ptr`. Таким образом, можно даже не писать велосипед

```

1 template <typename U>
2 int foo (int x) {
3     boost::scoped_ptr<U> ptr(new U(x));
4     // .... some code ....
5     if (some condition) {
6         return -1;
7     }
8     // ....
9     return 0;
10 }

```

Отдельный вопрос: почему ничего такого не было включено в стандарт, но он получит свой ответ далее. Итак, простейшая проблема, таким образом решена. Пора перейти к более сложным случаям. А конкретно: к случаям, когда указатель вынужден утекать из области видимости.

### 5.3.2 Автоматические указатели и их проблемы

Иногда указатели могут “утекать” за scope. Это вполне естественно и нормально и логично иметь нормальные способы это обработать. Пример, приведённый ранее может быть расширен следующим образом:

```

1 template <typename T> bar(T*);
2
3 template <typename U>
4 int foo (int x) {
5     boost::scoped_ptr<U> ptr(new U(x));
6     // .... some code ....
7     if (some condition) {
8         bar<U>(ptr.get());
9     }
10    // ....
11    return 0;
12 }
```

Увы, это очень плохая идея. В строчке вызова функции `bar` утекает сырой указатель. Теперь мы совсем не можем быть уверены, что его никто не освободит там, куда он утёк, и уж точно никак не можем это проконтролировать.

Древним выходом из этой ситуации является `auto_ptr`. Введённый в 98 стандарте, этот указатель определял конструктор копирования через передачу владения ресурсом

```

1 ARAIIPtr(ARAIIPtr& rhs) : ptr_(rhs.ptr_) {
2     rhs.ptr_ = nullptr;
3 }
4 ARAIIPtr& operator= (ARAIIPtr rhs) {
5     swap(*this, rhs); return *this;
6 }
```

Такой подход позволяет эффективно решить проблему: теперь нет нужды позволять утекать голому указателю, он передаётся вместе с владением

```

1 template <typename T> bar(auto_ptr<T>);
2
3 template <typename U>
4 int foo (int x) {
5     auto_ptr<U> ptr(new U(x));
```

```

6   // .... some code ....
7   if (some condition) {
8     bar<U>(ptr);
9   }
10  // ....
11  return 0;
12 }
```

Увы, передача владения таким образом подразумевает, в некотором смысле **слишком** тихое копирование. Типичный случай: пусть есть класс, который делает внутри себя резервную копию своего аргумента.

```

1 template <typename T> struct Brittle {
2   T working_, reserve_;
3   Brittle (T val) : working_(val), reserve_(working_) {}
4 };
```

Попытка использовать этот класс, параметризованный `auto_ptr` приведёт к удивительным и разрушительным последствиям.

```
1 Brittle<auto_ptr<int>> b (auto_ptr<int>(new int(42)));
```

Очень плохо то, что это скомпилируется без каких-нибудь ошибок, управление логично и закономерно утечёт в резервную копию, рабочая копия останется пустой и потом, где-нибудь это выстрелит.

Но самым-самым плохим является то, что приведённый выше класс `Brittle` очень напоминает неинтрезивные контейнеры стандартной библиотеки, которые по сути делают всё то же самое, только ещё сложнее.

Любые попытки использовать `auto_ptr` со стандартными контейнерами опасны и порочны. Специальное сокращение COAP расшифровывающееся как “container of auto ptrs” долгое время было синонимом таящегося и абсолютного зла.

При очевидной полезности `auto_ptr`, поражает воображение его способность разрушать и приводить в негодность самые невинно выглядящие контексты. Никто не заподозрил бы экземпляр COAP в приведённом выше классе `Brittle`, он же совсем не похож на классические контейнеры.

Отчаяние и безысходность продолжались до 2011 года, когда стандарт был наконец пересмотрен в сторону введения так называемой семантики перемещения (см. 3.5.4), которая, в свою очередь, позволила ввести настоящую абстракцию я уникального владения ресурсом – `unique pointers`.

### 5.3.3 Уникальное владение

*I'm the one that has to die  
when it's time for me to die*

– Jimi Hendrix

Если вдуматься, основная проблема использования копирования для передачи владения ресурсом – не в самой передаче владения, а именно в том, что для этого используется копирование, которое просто семантически для этого не предназначено.

Соответственно с появлением перемещающего присваивания, естественный выход из ситуации это запретить копирование, но открыть перемещение.

```

1 URAIIPtr(const URAIIPtr&) = delete;
2 URAIIPtr& operator=(const URAIIPtr&) = delete;
3 URAIIPtr(URAIIPtr&& rhs) : ptr_(rhs.ptr_) {
4     rhs.ptr_ = nullptr;
5 }
6 URAIIPtr& operator=(URAIIPtr&& rhs) {
7     swap(*this, rhs); return *this;
8 }
```

Эта идея позволяет решить проблему передачи ресурса за пределы области видимости.

```

1 template <typename T> bar(unique_ptr<T>);
2
3 template <typename U>
4 int foo (int x) {
5     auto ptr = make_unique<U>(x);
6     // .... some code ....
7     if (some condition) {
8         bar<U>(move(ptr));
9     }
10    // ....
11    return 0;
12 }
```

Здесь следует в первую очередь обратить внимание на использование `std::move`. Теперь передача ресурса это действительно и явно перемещающая передача.

А что насчёт `Brittle` – может тут спросить внимательный читатель.

```
1 Brittle<unique_ptr<int>> b (unique_ptr<int>(new int(42)));
```

Разумеется коварная строчка из предыдущего раздела даже не скомпилируется.

Также стоит обратить внимание на метод создания указателя. Тут есть два варианта: классический через конструктор и использованный выше.

```
1 auto ptr = make_unique<U>(x);
2 unique_ptr<U> ptr (new U(x));
```

В данном случае они посимвольно равны в записи. Тем не менее, разница есть и кроется в основном в вопросах безопасности исключений. Если `new` отработало, а потом конструктор `unique_ptr` бросил исключение, то выделенная память утечёт.

Единственное, что сначала кажется неудобным это аллокация массива. Кажется, что в конструкторе можно написать

```
1 unique_ptr<int> ptr (new int[1000]()); // ERROR
```

Тогда как `make_unique` этого не позволяет. Но это, пожалуй, аргумент за `make_unique`. Потому что такая запись является грубой ошибкой. И действительно: тут смешаны формы `new[]` и `delete`.

Есть два варианта корректно аллоцировать массив и оба используют специализацию шаблона для массивов.

```
1 auto ui = make_unique<int[]>(1000);
2 unique_ptr<int[]> ui (new int[1000]);
```

Теперь правильно спарены `new[]` и `delete[]`. Если записать такую схему в велосипеде `URAIIPtr`, она будет выглядеть следующим образом.

Во-первых удалитель теперь будет параметром шаблона

```
1 template <typename T, typename Deleter = default_delete<T>>
2 class URAIIPtr {
3     T *ptr_;
4     Deleter del_;
5 public:
6     URAIIPtr(T *ptr = nullptr, Deleter del = Deleter()) :
7         ptr_(ptr), del_(del) {}
8     ~URAIIPtr() { del_(ptr_); }
```

Во-вторых он будет частично специализирован

```

1 template <typename T> struct default_delete {
2     void operator() (T *ptr) { delete ptr; }
3 };
4
5 template <typename T> struct default_delete<T[]> {
6     void operator() (T *ptr) { delete [] ptr; }
7 };

```

Разумеется, в такой системе кажется, что есть третий выход: воспользоваться своим удалителем.

**Вопрос к студентам:** напишите свой удалитель к некорректному варианту, использующему `unique_ptr<int>` и при этом `new int[1000]()` в конструкторе так, чтобы этот вариант заработал.

Пользовательский удалитель может быть сколь угодно сложным: класс с нетривиальным конструктором, лямбда с захватом контекста. Но чаще всего нужны старые добрые синхронные функции.

```

1 Resource *create();
2 void destroy(Resource *);
3 unique_ptr<Resource, decltype(&destroy)> ures (create(),
    destroy);

```

Увы, в этом случае пользователь платит штраф на размер указателя, который теперь вынужденно включает указатель на функцию.

**Вопрос к студентам:** что вы думаете про `unique_ptr<void>`, можно ли его использовать как уникальный обобщённый указатель?

Очень часто люди, узнав про `unique_ptr` радуются (правильно радуются) и заменяют им голые указатели члены в своих классах (правильно заменяют). Но при этом часто возникает проблема с предварительными объявлениями.

```

1 class MyClass; // forward declared

```

В этом случае старый вариант не составляет проблем.

```

1 struct MyWrapper {
2     MyClass *c;
3     MyWrapper() : c(nullptr) {};
4 };

```

В то же время новый вариант не компилируется.

```

1 // FAIL!
2 struct MySafeWrapper {
3     unique_ptr<MyClass> c;
4     MySafeWrapper() : c(nullptr) {};
5 };

```

Проблема состоит в том, что дефолтный удалитель просачивается в хедер и приносит туда вызов `delete`, а этот вызов для неполных типов не работает.

Разумеется проблема может быть решена кастомным удалителем

```

1 struct MyClassDelete {
2     void operator()(MyClass *p);
3 };
4
5 struct MySafeWrapper {
6     unique_ptr<MyClass, MyClassDelete> c;
7     MySafeWrapper() : c(nullptr) {}
8 };

```

**Вопрос к студентам:** видите ли вы иные способы решить проблему?

#### 5.3.4 Проектирование с уникальными указателями

Идиома PImpl уже рассматривалась в связи с исключениями и является хорошим и заслуженным инструментом программиста, средством своего рода “тальванической развязки” интерфейса с реализацией, за которую мы иногда согласны пожертвовать один уровень косвенности при обращениях.

Не может ли эта идиома быть существенно улучшена с использованием умных указателей? Конечно может. Итак, допустим исходный код представляет классический PImpl.

```

1 class Ifacade {
2     CImpl *const impl_;
3 public:
4     Ifacade() : impl_(new CImpl) {}
5     // .... other methods
6 };

```

**Вопрос к студентам:** понимаете ли вы почему указатель лучше держать константным?

Переход от неё к уникальному владению довольно прост.

```

1 class Ifacade {
2     const unique_ptr<CImpl> impl_;
3 public:
4     Ifacade() : impl_(new CImpl) {}
5     // .... other methods
6 };

```

К слову можно упомянуть тот факт, что `const unique_ptr` это идеальный `scoped_ptr`. И действительно: он никак даже случайно не может быть ни скопирован ни (он же `const`) перемещён.

Что если класс управляет не одним, а более указателями. Например если это дерево? Дерево уникальных указателей выглядит крайне просто и естественно.

```

1 template <typename Data> class Tree {
2     struct Node {
3         unique_ptr<Node> left, right;
4         Data d;
5     };
6     unique_ptr<Node> top_;
7     void release_subtree(unique_ptr<Node> u) {}
8     // .... other 42 methods .....
9 };

```

Обратите внимание на реализацию метода `release_subtree`, она довольно симпатична. Почему это работает? Потому что сдвинутый туда уникальный указатель оказывается в этой области видимости и по выходе из неё уничтожается.

TODO: Про возможное переполнение стека при удалении и про решение

```

1 void release_subtree(unique_ptr<Node> u) {
2     // TODO: нетривиально решено
3 }

```

**Вопрос к студентам:** Можно ли (если да то как и если нет, то почему) сделать на `unique_ptr` направленный ациклический граф?

### 5.3.5 Совместное владение

*A shared pointer is as good as a global variable*

– Sean Parent

К сожалению, с деревом всё гладко лишь до тех пор, пока оно инкапсулировано. Но что если нужен метод, который ищет узел в дереве и отдаёт указатель на него внешнему владельцу? Не хотелось бы вместе с узлом передавать всё поддерево, исключая его из дерева. Также позволять утекать сырому указателю совсем не хочется.

Решением может быть: вернуть указатель с совместным владением.

```

1 template <typename Data> class Tree {
2     struct Node {
3         shared_ptr<Node> left, right;
4         Data d;
5     };
6     shared_ptr<Node> top_;
7 public:
8     shared_ptr<Node> find (int inorder_pos) {
9         shared_ptr<Node> spn;
10        // searching
11        return spn;
12    }

```

Что это за указатель с разделяемым владением?

Можно вспомнить основную проблему RAII: копирование. Пока что были рассмотрены первые три выхода, указатели с разделяемым использованием открывают четвёртый вариант.

1. Отказ от копирования: `scoped_ptr`, `const unique_ptr`
2. Копирование с передачей владения `auto_ptr`
3. Перемещение с передачей владения `unique_ptr`
4. Копирование с подсчётом ссылок и удалением при обнулении счётчика `shared_ptr`

Принципиальная схема разделяемого указателя приведена на (рис. 5.12).

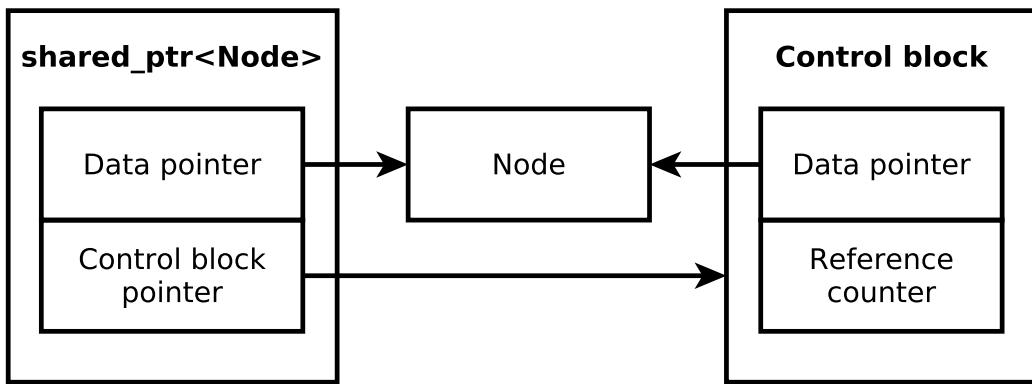


Рис. 5.12: Разделяемые указатели: контрольный блок

Этой картинке не стоит чрезмерно верить, её уточнение – дело ближайшего времени. Основные элементы картинки это данные, которые сохраняются в куче пока счётчик ненулевой, контрольный блок, который выделен отдельно и содержит счётчик и указатель на данные и конкретные разделяемые указатели, которые ссылаются на контрольный блок.

Существует (начиная с C++14) два способа создать разделяемый указатель.

1. С помощью конструктора

```
1 shared_ptr<Node> p2(new Node());
```

2. С помощью специальной функции `make_shared`

```
1 shared_ptr<Node> p1 = make_shared<Node>();
```

В первом случае данные и контрольный блок создаются двумя разными выделениями и существуют отдельно, как это и показано на (рис. 5.12). Этот вариант менее эффективен и менее безопасен относительно исключений, чем второй вариант, когда данные и контрольный блок создаются одним выделением динамической памяти (см. рис. 5.13).

Метод через `make_shared` не сработает если в классе есть закрытые или защищённые конструкторы.

```
1 class A {
2     int val;
3     A(int v): val(v){}
4 public:
```

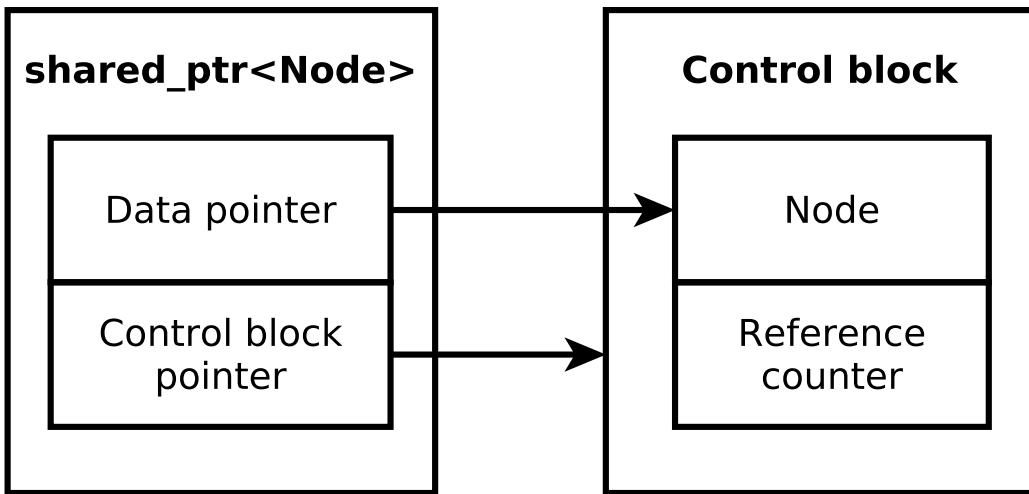


Рис. 5.13: Устройство разделяемого указателя после make\_shared

```

5     shared_ptr<A> createNext(){
6         return make_shared<A>(val); // FAIL
7     }
8     shared_ptr<A> createNext(){
9         return shared_ptr<A>(new A(val)); } // OK
10    };
11 // .....
  
```

Проблема очевидна: `make_shared` не друг класса и не имеет доступа к его непубличной части.

Возвращаясь к проблеме метода `find`, можно заметить, что он возвращает слишком много. Что если нужна не вся нода, а только указатель на данные из неё и всё-таки не хочется делать его сырым указателем?

```

1 template <typename Data> class Tree {
2     struct Node {
3         shared_ptr<Node> left, right;
4         Data d;
5     };
6     shared_ptr<Node> top_;
7 public:
8     shared_ptr<Data> find (int inorder_pos) {
9         shared_ptr<Node> spn;
10        // .... searching ....
11        return {spn, &(spn->d)};
  
```

12      }

Этот способ создания называется созданием с алиасингом (aliasing constructor) и проиллюстрирован на (рис. 5.14)

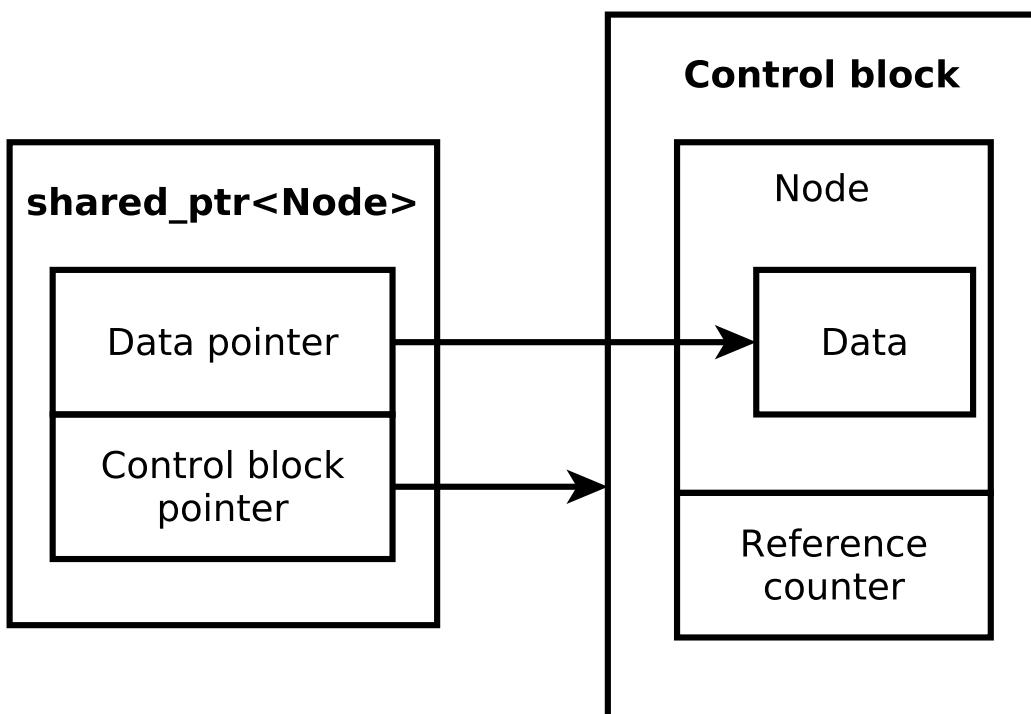


Рис. 5.14: Алиасинг в разделяемых указателях

Благодаря созданию с алиасингом, указатель может ссылаться на часть данных, управляемых его контрольным блоком, а не на все данные.

Разумеется, само наличие контрольного блока (подозрительно напоминающего похороненную было корову, см. 5.1.4) может стать проблемой *per se*. Например может быть такое, что два разных разделяемых указателя были сделаны из одного.

```

1 Node *n = new Node();
2 shared_ptr<Node> spn1(n);
3 shared_ptr<Node> spn2(n);
  
```

Это приводит к созданию двух контрольных блоков и той же проблеме двойного удаления, от которой исходно пытались уйти с помощью подсчёта ссылок

И в общем такие ситуации это ещё один аргумент за `make_shared`, который исключает эти проблемы.

Тем не менее, редко попадаясь на приведённый выше абсурдный сценарий, люди часто попадаются на него же, но когда указателем является `this`. Следующий фрагмент кода куда сложнее выкупить.

```

1 struct Node {
2     shared_ptr<Node> getspn() {
3         return shared_ptr<Node>(this); // ouch
4     }
5 };
6
7 shared_ptr<Node> bp1 = make_shared<Node>();
8 shared_ptr<Node> bp2 = bp1->getspn();
```

Правильная стратегия действий использует CRTP (см. 4.4). В класс, который хочет экспортировать указатель на самого себя подмешивается шаблонная добавка `enable_shared_from_this`, добавляющая в класс специальный метод `shared_from_this()`

```

1 struct Node: enable_shared_from_this<Node> {
2     shared_ptr<Node> getspn() {
3         return shared_from_this();
4     }
5 };
```

О том как это работает говорить пока рано, так как используется механизм слабых указателей, (см. 5.3.6)

```

1 Node n; // shall be owned
2 shared_ptr<Node> gp1 = n.getptr(); // UB
```

Впрочем это UB остаётся UB только до C++17. Начиная с 2017 года это выбрасывает исключение `std::bad_weak_ptr`.

Обычные указатели ковариантны и это довольно удобно, так как между ними можно статически переходить.

```

1 class A {};
2 class B : public A {};
3 B *b = new B();
4 A *a = static_cast<A*>(b);
```

Чтобы сохранить это свойство, статическое преобразование было сделано специально для разделяемых указателей (название `static_pointer_cast` не слишком удачное, так как он не действует для `unique_ptr` и прочих).

```
1 shared_ptr<B> b = make_shared<B>();
2 shared_ptr<A> a = static_pointer_cast<A>(b);
```

Аналогично работают `dynamic_pointer_cast` и `const_pointer_cast`, имитируя известные приведения.

TODO: пользовательские удалители

### 5.3.6 Слабые указатели

Главная проблема указателей с совместным владением это возможность циркулярных ссылок. Следующий код выглядит безобидно, но создаёт утечку памяти.

```
1 struct Node {
2     shared_ptr<Node> parent, left, right;
3 };
4
5 {
6     shared_ptr<Node> master = make_shared<Node>();
7     shared_ptr<Node> slave = make_shared<Node>();
8     slave.parent = master;
9     master.left = slave;
10 } // LEAK
```

Давайте разберём в деталях что тут происходит. Сначала создаётся указатель на `master` и его счётчик ссылок равен единице. Далее то же самое происходит для `slave`. Далее `slave` устанавливает совместный указатель на родителя, используя для этого существующий `master`. Количество ссылок в контрольном блоке `master` становится равно двум. Далее `master` ставит `slave` своим левым потомком, увеличивая счётчик ссылок в его контрольном блоке до двух.

Далее самое интересное. Завершается область видимости. Что происходит? Объекты `master` и `slave` выходят из зоны видимости, их счётчики уменьшаются на единицу. Но сами они от этого не умирают, так как кросс-ссылки на контрольные блоки продолжают держать их живыми.

И это не какие-то баги в реализации. Всё дело в отношении владения. Когда оно взаимное, оно создаёт крайне живучую конструкцию.

Выход из ситуации: заменить указатель наверх на невладеющий указатель.

```

1 struct Node {
2     weak_ptr<Node> parent
3     shared_ptr<Node> left, right;
4 };

```

Идея слабого указателя и получающийся при этом вид контрольного блока показаны на (рис. 5.15). Здесь в контрольном блоке пропущены пользовательские удалители (которых в данном случае в общем и нет), зато показан так называемый счётчик слабых ссылок.

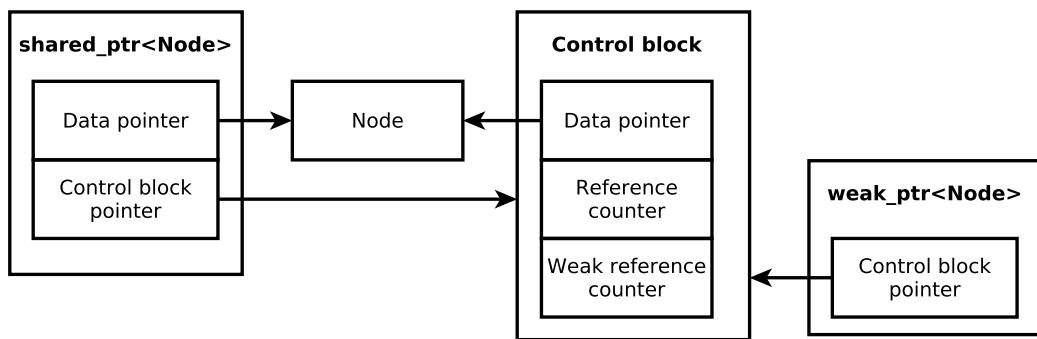


Рис. 5.15: Слабые указатели и устройство контрольного блока

При рассматривании рисунка возникает закономерная мысль: почему слабый указатель хранит только указатель на контрольный блок? Правильный ответ: потому что его **нельзя разыменовать**.

```

1 auto t = make_shared<int>(42);
2 weak_ptr<int> w = t;
3 int xt = *t; // OK
4 int xw = *w; // CE

```

Всё, что можно сделать со слабым указателем это атомарно “зашёлкнуть” его, превратив в сильный и только потом разыменовать.

```

1 auto tprime = w.lock();
2 int xtp = *tprime;

```

Зашёлкивание не вовремя может создать те же проблемы циклической ссылки, но они более контролируемы. Например временно зашёлкнуть указатель `parent` у структуры `Node` приведённой выше – практически безопасно.

Тем не менее, наличие слабых ссылок выявляет неожиданный аргумент против `make_shared`. Идея показана на (рис. 5.16).

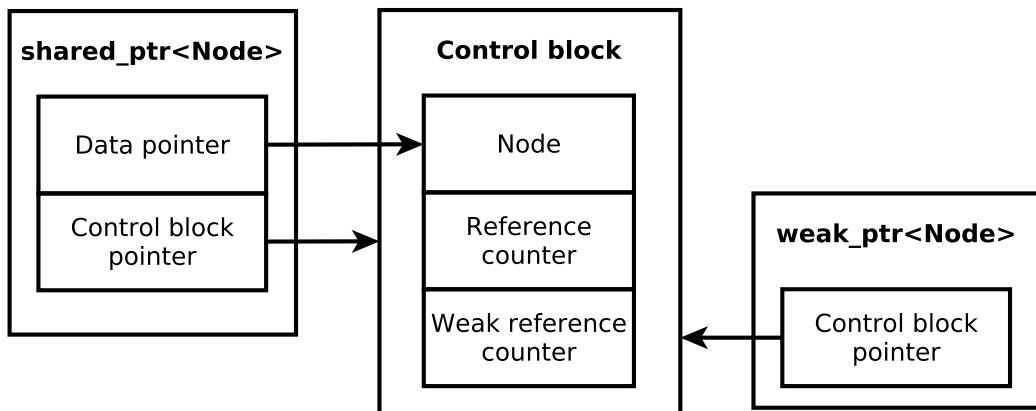


Рис. 5.16: Проблема `make_shared` и слабых ссылок

Здесь блок был выделен с помощью одного выделения вместе с данными и на него есть и сильные и слабые ссылки. Если на блок не останется ни одной сильной ссылки, то что произойдёт?

- Control block живёт пока есть слабые ссылки
- Data освобождается
- Data pointer становится `nullptr`
- Если weak pointer указывает на control block без сильных ссылок, то он expired

Интересный термин `expired` (истёк) требует разбора. Обычный указатель может повиснуть если то, на что он указывал было удалено. Но слабый указатель не провисает. Он истекает и это можно детерминированно проверить.

```

1 weak_ptr<int>
2 foo ()
3 {
4     auto res = make_shared<int>(42);
5     return res;
6 } // expiration
7
8 weak_ptr<int> result = foo();
  
```

```
9 assert (result.expired());  
10 assert (result.lock() == nullptr);
```

Классический пример проектирования со слабыми указателями это кэш Майерса [7].

```
1 template <typename T> shared_ptr<T>  
2 getObject(int id)  
3 {  
4     static unordered_map<int, weak_ptr<T>> cache;  
5     auto result = cache[id].lock();  
6     if (!result)  
7         cache[id] = result = createObject<T>(id);  
8     return result;  
9 }
```

Здесь функция `createObject` создаёт объекты и кэширует их в неупорядоченном ассоциативном массиве. Пользователь получает разделяемый указатель и пока он им пользуется объект жив, а когда он его удаляет, в кэше остаётся истёкший слабый указатель. Это позволяет максимально экономно расходовать память и детерминированно контролировать подвисания указателей.

## 5.4 Последовательные контейнеры

*When choosing a container, remember vector is best  
Leave a comment to explain if you choose from the rest*

*– Tony Van Eerd*

Контейнеры STL отражают абстракцию структур данных. Все практически важные структуры уже представлены или могут быть представлены контейнерами. Наиболее известны следующие последовательные контейнеры:

- **vector** – массив с гарантиями непрерывной области памяти и динамическим ростом с одной стороны
- **deque** – массив со странично организованной памятью и ростом с двух сторон.
- **list** – двусвязный список с быстрой вставкой и удалением элементов
- **forward\_list** – односвязный список с быстрой вставкой и удалением элементов
- **array** – массив с фиксированным на этапе компиляции размером

Кроме того, три известных контейнерных адаптора чаще всего используются именно с последовательными контейнерами и, поэтому, тоже имеют отношение к этому разделу.

- **stack** – стек, обычно сделанный на базе **deque**
- **queue** – очередь, обычной сделанная на базе **deque**
- **priority\_queue** – очередь с приоритетами, обычно сделанная на базе **vector**

Кроме них существуют также ассоциативные контейнеры, которые будут рассмотрены в (5.9).

### 5.4.1 Возможности контейнеров и требования к элементам

Последовательные контейнеры STL поддерживают общие операции:

- **empty** – проверка пустоты контейнера
- **max\_size** – максимальный размер в текущей реализации
- **swap** – обмен контейнерных переменных содержимым
- **size** – действительный размер контейнера (кроме **array**, где размер – часть типа)
- **clear** – очистка контейнера (кроме **array**, там очистка невозможна)
- **front** – первый элемент (также для всех кроме **forward\_list** поддержан метод **back**, возвращающий последний элемент)
- **begin, end, cbegin, cend** – получение итераторов (см. далее)

Любой контейнер содержит конструктор по умолчанию, копирующий конструктор и открытый невиртуальный деструктор. Последнее крайне важно. Стандарт (TODO: пункт?), нормы коммьюнити, а иногда даже и сообщения компилятора прямо и недвусмысленно **запрещают** наследование от шаблонных контейнеров стандартной библиотеки. Как мы помним из 3.6.5, наличие невиртуального деструктора в классе – достаточно основание никогда от него не наследовать.

Начиная с C++11 все они, казалось бы, должны быть отмечены как **final**. Увы, жизнь жестче. Никаких ограничений на наследование наложено не было и хуже того, сам товарищ Строструп в выложенной публично <https://isocpp.org/files/papers/4-Tour-Algo-draft.pdf> статье (являющейся частью четвертого издания его книги по C++) использует наследование от вектора.

```
1 template <typename T>
2 class Vec : public std::vector<T> {
3 public:
4     using vector<T>::vector; // ctors from vector, see 20.3.5.1
5
6     T& operator[](int i) { return vector<T>::at(i); }
7     const T& operator[](int i) const { return vector<T>::at(i); }
8 };
```

Это безумие, не только потому что перестанет работать удаление по указателю на базовый класс, а ещё и потому, что оператор индексации не перегружен, а перекрыт и это ведёт к неочевидному разрешению имён в полиморфных контекстах. Но это разрешено стандартом и, хм, это делает сам создатель языка.

Возможно это я чего-то не понимаю.

Контейнеры STL являются шаблонами, а все шаблонные интерфейсы являются неявными. Поэтому программист должен хорошо знать неявные требования, предъявляемые к элементам контейнеров, как к чему-то, что может быть помещено в данный контейнер.

- **Копируемость** – у объекта должен быть определён стандартный или пользовательский копирующий конструктор, который должен действительно копировать объект
- **Изменяемость** – объект должен быть lvalue с действительным оператором присваивания
- **Конструируемость** – хотя бы один из конструкторов и хотя бы один из деструкторов объекта должны быть открытыми

Некоторые контейнеры определяют дополнительные неявные требования, такие как наличие критерия сортировки (для упорядоченных конструкций) или конструктора по умолчанию.

Все контейнеры в стандартной библиотеке являются **неинтрузивными**. Это означает, что они хранят копии элементов, а не сами элементы. Также это означает, что конкретный элемент никак не может сказать не только хранится он сейчас в контейнере или живёт отдельно, но и **может ли он храниться в том или ином контейнере**.

### 5.4.2 От встроенных массивов к векторам

Одной из главных проблем, которые красной нитью проходили через все лекции была проблема управления памятью. Правильно спаривать `new` и `delete`, использовать RAII для управления ресурсами, управлять памятью в безопасном для исключений стиле – всё это важные и нужные навыки. Но внутри стандартной библиотеки содержится контейнер, который для большинства практических случаев позволяет в принципе избегать явного использования динамической памяти. Это класс `vector`. Он определён в заголовке `<vector>` стандартной библиотеки как:

```

1 template <typename T, typename Allocator = allocator<T> >
2 class vector;
```

Таким образом вместо

```

1 int *n = new int[10];
2 n[5] = 5;
```

Можно писать

```

1 vector<int> n(10);
2 n[5] = 5;
```

Для достижения такого эффекта, вектор использует переопределение оператора квадратных скобок, см. (3.3.4).

Но идентичность конструкций это ещё не всё. Вектора на самом деле лучше динамически распределяемых массивов. Скажем, в случае с явным выделением памяти, нет надежного способа стереть крайний элемент, для добавления каждого нового элемента нужна дорогая реаллокация, для резерва памяти нужен размещающий `new` и после всех этих плясок очень сложно сказать сколько же теперь места занимает такой массив в памяти. В случае вектора все эти проблемы решаются на удивление просто:

```

1 vector<int> v(10);
2 v[5] = 5;
3 size_t vsz = v.size();
4 v.pop_back();
5 if (v.empty()) { /* do something */ }
```

Кроме того, в случае явного размещения вы должны не забыть про освобождение памяти, а в случае вектора этим займется деструктор. Поэтому использовать вектор всегда предпочтительней, чем работать с массивами напрямую и с точки зрения безопасности исключений. Следующий пример представляет собой не слишком безопасный относительно исключений код:

```

1 template <typename T>
2 T foo (int n)
3 {
4     T *t = new T[n], ret = 0;
5
6     init_t (t, n); // exc!
7 }
```

```

8   for (int i = 0; i < n; ++i)
9   {
10     if (!check_legal(t[i])) // exc!
11     {
12       delete [] t;
13       throw std::runtime_error("Illegal element");
14     }
15
16     ret += t[i]; // exc!
17   }
18
19   delete [] t;
20
21   return ret;
22 }
```

Он изобилует проблемами и местами, где возможна утечка. К тому же его элементарно сложно читать. Давайте посмотрим как этот код магически трансформируется в нечто гораздо более прозрачное и безопасное.

```

1 template <typename T>
2 T foo (int n)
3 {
4   std::vector<T> t(n);
5   T ret = 0;
6
7   /* legal! vector in memory laid out continuously */
8   init_t (&t[0], n);
9
10  size_t vsz = t.size();
11  for (int i = 0; i != vsz; ++i)
12  {
13    if (!check_legal(t[i]))
14      throw std::runtime_error("Illegal element");
15
16    ret += t[i];
17  }
18
19  return ret;
20 }
```

При выбросе исключения, память под вектором будет корректно освобождена. Однако такое управление памятью, вовсе не означает, что программист не должен думать о памяти под вектором или что эта память не имеет значения.

Вектора проектировались с учётом необходимой совместимости и удовлетворяют строжайшей гарантии: память под ними всегда лежит одним непрерывным куском. Поэтому, например, если есть функция `init`, ожидающая указателя на непрерывную область данных

```
1 template <typename T>
2 void init (T*, int);
```

Почти всегда легально передать туда указатель на первый элемент вектора:

```
1 std::vector<T> t(n);
2
3 /* legal! vector in memory laid out continuously */
4 init_t (&t[0], n);
```

Это происходит от того, что почти все вектора отвечают двум условиям:

- Память в контейнере непрерывна
- Память организована линейно

Почти. Кроме одного уродливого исключения и это исключение – вектор булевых переменных.

Самое плохое в `vector<bool>`, что он не `vector` и что он не хранит `bool`.

Интенция при этом была хорошей. `vector<bool>` — это специализация шаблона `vector`, компактно хранящая данные, выделяя по одному биту на каждый булев признак, а не по одному байту, который традиционно занимает тип `bool`.

Увы, с таким подходом этот контейнер не хранит `bool`, потому что при индексации такой вектор не может вернуть ссылку на `bool`. По сути ему нужно каким-то образом вернуть ссылку на один бит. В результате приходится использовать `vector<bool>::reference` — некий проксирующий тип, который пытается имитировать своим поведением ссылку на `bool`.

И кроме того, с таким подходом он не является вектором, так как не удовлетворяет требованиям к контейнеру `vector` по линейной организации памяти:

```

1 vector<T> t(n);
2 /* illegal only if T is bool */
3 bool *b = &t[0];
4 assert (b[1] == t[1]);
```

В этом случае следующий за первым при инкременте указателя элемент вовсе не является тем элементом, который реально хранится на соответствующей позиции в контейнере.

Пример с вектором булевых значений крайне поучителен: при проектировании обобщенного кода такого рода специализации вредны. Лучше было бы сделать отдельный класс, такой как `std::bitvector`.

Допустим, теперь не осталось никаких сомнений, что память под вектором булевых переменных организована некорректно. Но всё ли на данный момент ясно с корректными случаями? Как ни странно, несмотря на то, что вектора сделаны для того, чтобы абстрагировать работу с памятью, забывать о памяти при работе с ними не получится.

Например, вектора в C++ поддерживают операцию `push_back`, для динамического роста. Поэтому новички часто пишут код вроде такого:

```

1 vector<int> v;
2 for (int i = 0; i != N; ++i)
3     v.push_back(i);
```

**Вопрос к студентам:** что не так в этом коде?

К сожалению, это убивает детерминированность в работе вектора с памятью и ведёт к  $O(\log(N))$  её полных перераспределений, поэтому аккуратный программист всегда использует резервирование памяти для того, чтобы исключить перераспределение при добавлении элементов

```

1 vector<int> v;
2 v.reserve(N); /* place for N ints */
3 for (int i = 0; i != N; ++i)
4     v.push_back(i);
```

Таким образом у вектора есть два независимых параметра: его ёмкость (`capacity`) и его размер (`size`). Изменение ёмкости аналогично выделению сырой памяти. Если ёмкость совпадает с размером и в вектор

что-то записывается, емкость увеличивается. Большинство реализаций каждую реаллокацию увеличивают свою ёмкость вдвое.

В частности, это означает, что вектор тяжелых объектов является плохой идеей без очень педантичного управления памятью, поскольку реальная память под ним может быть вдвое больше необходимой.

Кроме того, вектора, как и большинство контейнеров с разрешённой вставкой, разрешают также размещение (*emplace*) в них элементов. Рассмотрение этой возможности, которая уже упоминалась при разговоре про правые ссылки (см. 4.5.4) будет отложено до дек, которые в этом вопросе несколько симметричней (см. 5.4.6).

### 5.4.3 От векторов обратно к массивам

Вектор – хорошая вещь, но объективно он тяжеловесен, а возможность динамически его переаллоцировать не всегда нужна. Все разработчики годами жили с C-style arrays. Начиная с 11-го года в C++ поддерживается стандартный контейнер `std::array` – максимально легковесное приближение к встроенному массиву. Размер такого массива задается на этапе компиляции и не может быть изменен, но при этом все операции с ним так же просты, как и с прочими контейнерами стандартной библиотеки.

Массив на стеке с фиксированным размером и `std::array` выглядят похоже и в объявлении и в использовании.

```
1 int s_array[10]; // C style
2 array<int, 10> arr; // C++ style
3
4 s_array[5] = 3;
5 arr[5] = 3;
```

Но есть и различия. Первое различие заключается в том, что обычные массивы могут деградировать к указателям и терять информацию о размере.

```
1 void trap (Animal* animals, size_t size);
2
3 Animal four_animals[4];
4 Animal five_animals[5];
5
6 trap (four_animals, 4);
```

```
7 trap (five_animals, 5);
```

Здесь два вызова `trap` это два вызова одной и той же функции.

В то же время `std::array` ни к чему не деградирует, а его размер является частью его типа.

```
1 template <size_t sz>
2 void trap (array<Animal, sz> animals);
3
4 array<Animal, 4> four_animals;
5 array<Animal, 5> five_animals;
6
7 trap (four_animals);
8 trap (five_animals);
```

Здесь два вызова `trap` выведут совсем разные типы и окажутся двумя разными функциями (строго говоря – двумя разными экземплярами шаблонной функции).

Хуже того, в этом примере массив пойдёт в функцию по значению. Конечно, `std::array` даёт гарантии непрерывности по памяти, так что старая добрая ловушка работает и для новых групп животных.

```
1 void trap (Animal* animals, size_t size);
2
3 array<Animal, 4> four_animals;
4 array<Animal, 5> five_animals;
5
6 trap (four_animals.data(), four_animals.size());
7 trap (five_animals.data(), five_animals.size());
```

Второе и крайне важное отличие имеет дело с ковариантностью типов относительно обобщения.

**Ковариантным** относительно отношения `R` называется такое преобразование типов, которое сохраняется при этом отношении. Например если `A` обобщает `B`, то `A*` обобщает `B*`. В то же время `array<A>` и `array<B>` являются типами, не имеющими друг с другом ничего общего. Такое преобразование называется **инвариантным**.

```
1 class Dog : public Animal {
2     // something doggy style
3 };
4
```

```

5 void trap (Animal* animals, size_t size);
6
7 Dog dogs[5];
8 trap (dogs, 5); // ok

```

Собака это животное, поэтому ловушка в старом стиле работает для массива собак. Ловушка в новом стиле просто не поймёт в чём дело.

```

1 template <size_t sz>
2 void trap (array<Animal, sz> animals);
3
4 array<Dog, 5> dogs;
5 trap<5> (dogs); // error, array<Dog> is not array<Animal>

```

Шаблоны контейнеров не просто не ковариантны по прихоти комитета, этому на самом деле есть крайне разумное объяснение. Представьте они были бы ковариантны. Тогда следующий код имел бы место быть.

```

1 vector<Cat*> v1;
2 vector<Animal*>& v2 = v1; // ok, if covariant
3 v2.push_back(new Dog); // ouch

```

Все эти рассуждения на самом деле не относятся к одним лишь массивам, пронизывая весь язык. Ковариантность (без введения термина) для возвращаемых значений методов при перегрузке уже рассматривалась (см. 3.7.6). Подробнее всё это будет ещё раз уложено при разговоре про умные указатели (см. 5.3).

Продолжая разговор о преимуществах агрэйсов, можно заметить, что из них получаются симпатичные двумерные массивы.

```

1 template <typename T, int M, int N>
2 using array2d = std::array<std::array<T, N>, M>;

```

**Вопрос к студентам:** возможно ли построить многомерный массив, чтобы пользоваться им как-нибудь вот так:

```

1 typename MultiDimArray<float, 3, 4, 5, 6, 7>::type floats;

```

#### 5.4.4 Списки инициализации

Стандартный способ инициализации массивов в С действительно удобен. А вот для векторов в C++98 – так себе.

```

1 int a[7] = {2, 3, 5, 7, 9, 11, 13};
2 vector<int> b(10); /* = {...} ? */
3 a.push_back(1);
4 a.push_back(2);
5 /* ... ok, I'm tired already ... */

```

Начиная с C++11 то же самое доступно и для векторов и вообще для всех распространённых контейнеров.

```

1 int a[7] = {2, 3, 5, 7, 9, 11, 13};
2 vector<int> b(7) {2, 3, 5, 7, 9, 11, 13}; // C++11
3 vector<int> b(7) = {2, 3, 5, 7, 9, 11, 13}; // C++11

```

Этот механизм очень хорош, очень удобен и работает из коробки. Но чтобы пользоваться им не попадая в неприятности и иметь возможность делать то же для своих классов, необходимо хорошо представлять его внутренние приводные ремни.

На самом деле в C++ начиная с 2011 года есть два основных механизма инициализации с помощью фигурных скобок (напрашивается неологизм “фигурная инициализация” по аналогии с английским вариантом “braced initialization”, но автор склонен удержаться от столь необузданного словотворчества).

### Механизм 1. Расширенный синтаксис

Начиная с языка С встроенные структуры имели возможность инициализироваться достаточно симпатично

```

1 struct A {
2     int a_, b_;
3 };
4
5 A a = {1, 2};

```

Начиная с C++11 это доступно для любых классов с конструкторами, наряду с обычным вызовом конструкторов с круглыми скобками.

```

1 class B {
2     int a_, b_;
3 public:
4     B (int a, int b = 0) : a_(a), b_(b) {}
5 };
6
7 B b(1, 2); // C++98

```

```
8 B b{1, 2}; // C++11
```

К тому же, такой синтаксис защищает от неявных преобразований

```
1 B b(3.14); // ok, implicit cast: double -> int
2 B b{3}, c(3); // ok, the same ctor
3 B b{3.14}; // error, no implicit cast
```

## Механизм 2. Списки инициализации

И в том же году в том же стандарте была введена возможность писать конструкторы из списков инициализации.

```
1 class B {
2     int a_, b_;
3 public:
4     B (int a, int b = 0) : a_(a), b_(b) {}
5     B (std::initializer_list<int> il);
6 };
```

Теперь такая же строчка как раньше приведёт к вызову разных конструкторов.

```
1 B b{3}, c(3); // ok, but different ctors!
```

Видно, что два этих механизма несколько конфликтуют. Во всех конфликтах выигрывают списки инициализации. Проблема в том, что в случае вектора им есть у чего выигрывать. По стандарту у вектора есть конструктор, берущий два аргумента – сколько элементов выделить и чем их заполнить.

Поэтому очень похожий синтаксис инициализации может порождать разные контейнеры.

```
1 vector<int> v1 (3, 14); // [14, 14, 14]
2 vector<int> v2 {3, 14}; // [3, 14]
```

Можно ли адаптировать этот механизм для своих классов? Разумеется да и это даже несложно

```
1 template <typename T>
2 class Tree {
3     // .... some tree specific
4     bool add_node (T& data);
5 public:
6     Tree(initializer_list<T>& il) {
```

```

7     // .... here call add_node in loop
8 }
9 };

```

**Вопрос к студентам:** Список инициализации, как и вектор, как и массив – непрерывен в памяти. Но именно для списка инициализации, нет ли в этом решении каких-то, иногда ухудшающих его использование, недостатков?

#### 5.4.5 Первое представление об итераторах

Вместе классы `std::vector` и `std::array`, а также `std::initializer_list` составляют подмножество последовательных контейнеров с непрерывной памятью. Поэтому при работе с ними применялись обычные указатели. Но вообще при работе с контейнерами STL используются **итераторы**. Они будут очень подробно рассмотрены в (5.7). Тем не менее, некоторые базовые факты следует сообщить уже сейчас, иначе эта тележка не поедет.

Итак, было сказано, что при работе можно применять указатели и их можно применять.

```

1 vector<int> v(10);
2 auto pi = &v[0]; // --> pointer
3 pi += 3;
4 assert (*pi == v[4])

```

Но как узнать по указателю, что вектор закончился и следующий инкремент вынесет в пустоту? Никак. Зато это можно сделать с помощью итератора.

```

1 vector<int> v(10);
2 auto vi = v.begin(); // --> iterator
3 vi += 3;
4 assert (*vi == v[4]);
5 if (vi == v.end()) { /* something */ }

```

На (рис. 5.17) показано, что `v.end()` это специальный итератор за концом массива.

Пока что это всё, что нужно знать:

- Есть как минимум два способа получить итератор: методы `begin()` и `end()`

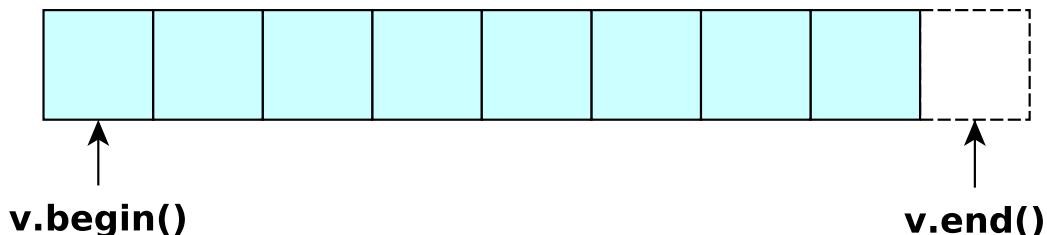


Рис. 5.17: Первое представление об итераторах

- Итератор ведёт себя как указатель: его можно разыменовать или инкрементировать
- Есть специальный ограничивающий итератор на конец контейнера.

Дерево из 5.4.4 с помощью итераторов может быть заполнено из списка инициализации довольно просто.

```

1 Tree(initializer_list<T> il) {
2     for (auto it = il.begin(); it != il.end(); ++it)
3         add_node(*it);
4 }
```

Дополнительная бонус-информация заключается в том, что большинство контейнеров поддерживают как метод `assign`, принимающий диапазон итераторов и перекидывающая в контейнер содержимое этого диапазона, так и специальный конструктор из двух итераторов, который делает то же самое но при создании. Ниже показано использование этого конструктора для перекидывания списка инициализации в вектор.

```

1 class Introw {
2     vector<int> v_;
3 public:
4     Introw(initializer_list<int> il) : v_(il.begin(), il.end())
5 }
```

В целом, итераторы делают многие вещи проще и естественней. Разговор о них будет продолжен позже, а пока следует поговорить о контейнерах, для которых они нужны – то есть таких, которые не поддерживают память одним куском и работу с ней через обычные указатели.

### 5.4.6 Деки

Непрерывность памяти это очень строгая гарантия. На самом деле, последовательный контейнер может быть устроен так, чтобы не обязательно её давать – например занимать произвольные участки памяти. Например это позволило бы сделать в таком контейнере эффективный `push_front`, `insert` или `emplace` после любого элемента и так далее. Для таких целей, библиотека поддерживает односвязные списки (`std::forward_list`), двусвязные списки (`std::list`) и деки (`std::deque`).

Дека (`deque`, от double ended queue, двусторонняя очередь, ноозвученная также с английским словом `deck` – колода карт) это почти магический контейнер, который поддерживает:

- Доступ к любому элементу за  $O(1)$
- Вставку в начало и в конец за  $O(1)$
- Вставку в произвольное место за  $O(n)$

**Вопрос к студентам:** как бы вы организовали такой контейнер?

Самое главное в деке – то, что она кусочно линейна и значит не требует резервирования памяти. Это тот самый контейнер, который **действительно** абстрагирует программиста от работы с памятью. Например следующий код уже встречался при разговоре о векторах

```
1 deque<int> v;
2 for (int i = 0; i != N; ++i)
3     v.push_back(i);
```

Но сейчас (и это кажется невероятным) с ним всё хорошо. Более того. Со следующим кодом, который для векторов был бы сущим безумием, тоже всё хорошо.

```
1 deque<int> v;
2 for (int i = 0; i != N; ++i)
3     v.push_front(i);
```

Здесь элементы добавляются в начало контейнера и это тоже происходит асимптотически быстро. Пожалуй единственный случай, когда программисту нужно думать о накладных расходах при работе с деками это накладные расходы на копирование тяжёлых объектов

```

1 struct Heavy {
2     explicit Heavy (int sz) {
3         cout << "Heavy created" << endl;
4     }
5
6     Heavy(const Heavy &rhs) {
7         cout << "Heavy copy-constructed" << endl;
8     }
9
10    Heavy (Heavy &&rhs) noexcept {
11        cout << "Heavy move-constructed" << endl;
12    }
13
14    Heavy& operator= (const Heavy &rhs) {
15        cout << "Heavy copied" << endl;
16        return *this;
17    }
18
19    Heavy& operator= (Heavy &&rhs) noexcept {
20        cout << "Heavy moved" << endl;
21        return *this;
22    }
23 };

```

Теперь становится прозрачным, что в случае использования `push_front` для помещения таких объектов в деку:

```

1 for (int i = 0; i != 5; ++i)
2     d.push_front(Heavy(i));

```

Каждый из них должен быть не только создан, но и перемещён. А в случае использования `emplace_front` – только создан.

```

1 for (int i = 0; i != 5; ++i)
2     d.emplace_front(i);

```

Более подробно эта идея обсуждается в разделе про правые ссылки, так как существенно использует этот механизм (см. 4.5.4).

Майерс [7] рекомендует рассмотреть использование деки в качестве вашего основного контейнера.

### 5.4.7 Списки

Классический список это двусвязный `std::list`, но начиная с C++11 добавлен также односвязный `std::forward_list`, который тратит вдвое меньше ценной памяти на поддержку указателей (ценой отсутствия итерируемости назад).

В целом со списками можно делать то же самое, что и с векторами или деками: инициализировать, обращаться по итератору, добавлять и удалять элементы. Для добавления элементов в любое место служит у двусвязных списков метод `insert`, который берёт итератор позиции сразу перед вставкой. Также удобно удалять элементы, это всегда происходит за  $O(1)$  и для этого есть специальный метод `remove`.

А вот переопределения квадратных скобок для списков не сделало, так как они бы внесли ненужную скрытую сложность. Увы, доступ к элементу по индексу в списке производится не менее чем за  $O(N)$ .

Самой интересной возможностью у списков является **сплайс**. Это техника, позволяющая асимптотически быстро вынуть кусок из одного контейнера и вставить его в другой. Она проиллюстрирована на (рис. 5.18).

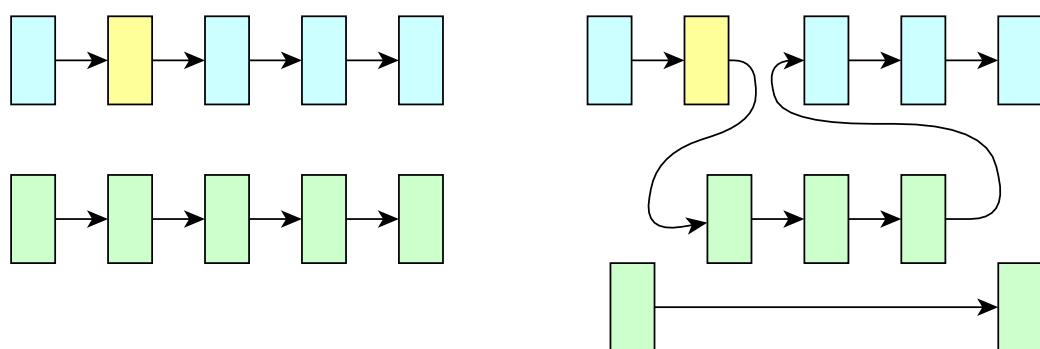


Рис. 5.18: Сплайс для односвязных списков

В качестве тренировки можно рассмотреть работу этой возможности. Первое, что надо сделать это подготовить сцену: обзавестись двумя списками

```

1 forward_list<int> first = { 1, 2, 3 };
2 forward_list<int> second = { 10, 20, 30 };
3 auto it = first.begin(); // it -> 1
```

Тут видно как работает инициализация списком и взятие итератора на первый элемент. Теперь можно перекинуть весь второй список в первый.

```
1 first.splice_after (first.before_begin (), second);
```

Оп. Теперь второй список пуст, а первый содержит элементы 10, 20, 30, 1, 2, 3

При этом итератор `it` остался тем же и так же указывает на элемент 1! В разделе про итераторы будет подробнее рассмотрена эта циникальная особенность списков: итераторы остаются **валидными** при гораздо большем спектре операций над списком, подробнее см. (5.7.6).

Далее можно, например, перекинуть элементы со второго по `it` в список `second`

```
1 second.splice_after (second.before_begin(), first, first.begin
    (), it);
```

Снова магия. Теперь первый список содержит 10, 1, 2, 3, а второй 20, 30

**Вопрос к студентам:** как теперь перекинуть все элементы второго списка начиная со второго (то есть в данном случае только 30) в первый список?

Тренироваться со сплайсом забавно, но у стандартных списков есть и тёмные стороны. Например они требуют некоторой аккуратности при необходимости вычислить размер.

```
1 template <typename Container>
2 void foo (Container &c)
3 {
4     if (c.size() == 0)
5     {
6         // processing
7     }
8     // processing
9 }
```

Этот обобщённый код выглядит хорошим только пока не предположить, что он должен работать со списками. В этом случае `size` работает не за константное, а за линейное время.

К счастью, все стандартные контейнеры поддерживают функцию `empty ()`, выполняющуюся за константное время. Поэтому правильная версия

вышеприведённого фрагмента кода будет выглядеть как-то так.

```
1 if (c.empty())
2 // processing
```

И тут может возникнуть закономерный вопрос: но почему это так? Казалось бы, размер вполне можно было хранить как атрибут и отдавать за  $O(1)$ .

Увы, здесь есть чисто инженерный баланс: если размер хранится как атрибут, то операция `splice` должна его обновлять, а для этого пересчитывать элементы которые она удаляет и вставляет. Но это сделает сложность `splice` линейной. Таким образом выбор стоит так: из пары методов `size` и `splice` для списков один любой может быть  $O(1)$ , но никогда не оба.

**Вопрос к студентам:** можете ли вы себе представить двусвязный список, который делает `reverse` за константное время? Чем придётся заплатить?

#### 5.4.8 Адаптеры

Некоторые распространённые “контейнеры”, такие как стек или очередь, на самом деле не являются контейнерами в смысле стандартной библиотеки, поскольку не определяют внутреннего представления своих данных. Что мы подразумеваем, говоря “стек”? Всего лишь набор операций, позволяющий снять элемент, положить элемент, посмотреть последний элемент. Такой набор операций может работать и для вектора и для деки и для списка. Поэтому в стандартной библиотеке, стеки и очереди реализованы не как контейнеры, а как **адаптеры** контейнеров.

На (рис. 5.19) показаны адаптеры стека и очереди. Контейнеры под ними в принципе произвольные, но стандарт даёт разумные умолчания.

Всего в стандартной библиотеке сейчас определены три вида адаптеров:

1. **stack** – LIFO стек над последовательным контейнером

```
1 template <class T, class Container = deque<T> >
2 class stack;
```

2. **queue** – FIFO очередь над последовательным контейнером

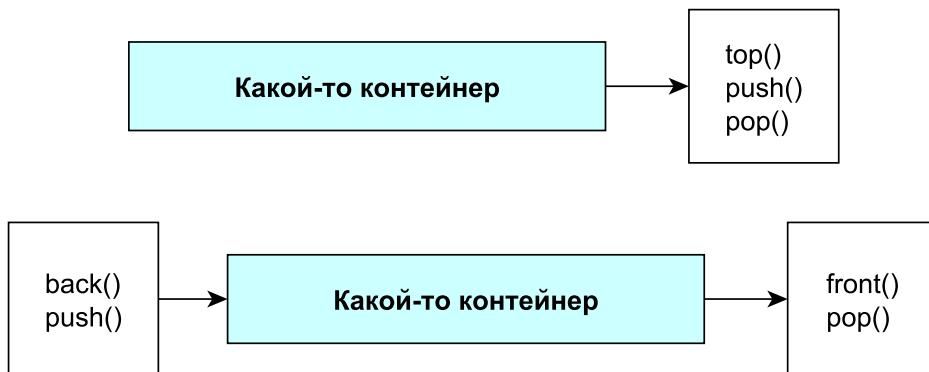


Рис. 5.19: Адаптеры стека и очереди

```

1 template <class T, class Container = deque<T> >
2 class queue;
3
3. priority_queue – очередь с приоритетами (как binary heap) над
последовательным контейнером
1 template <class T,
2         class Container = vector<T>,
3         class Compare = less<typename Container::
4           value_type>>
4 class priority_queue;

```

К сожалению адаптеры не слишком ортогональны: они берут одновременно и класс и тип, которым он параметризован. Это даёт неограниченные возможности выстрелить себе в ногу.

```

1 stack <int> s; // ok, stack <int, deque<int>>
2 stack <int, vector<long>> s1; // bad
3 stack <int, vector<char>> s2; // really bad
4 s2.push(1000);

```

**Вопрос к студентам:** что реально будет помещено в стек s2?

Казалось бы логично использовать шаблонные шаблонные параметры. Простейшая попытка такого рода может быть проиллюстрирована ниже

```

1 template <class T,
2         template <class T> class Container = deque>
3 class stack;

```

Казалось бы такой адаптер уже нельзя использовать неправильно.

```
1 stack<int, vector> s;
```

**Вопрос к студентам:** почему эта система не будет работать?

Кроме того, некоторые контейнеры просто не являются подходящими. Хуже всего, что никакой статической проверки не предусмотрено и неподходящий контейнера часто выясняется внезапно.

```
1 stack <int, forward_list<int>> s; // ok
```

Эта строчка превосходно скомпилируется, но попытка использовать любой метод из интерфейса будет провалена.

```
1 s.push(100); // fail: no push_back
2 s.pop(); // fail: no pop_back
3 s.top(); // fail: no back
```

Эти ошибки неочевидны. Стек вполне может быть сделан на односвязном списке.

**Вопрос к студентам:** вы понимаете как сделать стек на односвязном списке?

Но увы, адаптер `std::stack` требует (неявно требует) вполне определённый интерфейс.

#### 5.4.9 Контейнеро-подобные классы

В этом разделе речь пойдёт об удобных контейнеро-подобных классах, не являющихся, тем не менее, полноценными контейнерами. Начать стоит с мотивирующего примера.

Очень часто люди, привычные к математической нотации вектора неудоумеваются почему в C++ оператор сложения для векторов не работает в математическом смысле.

Например, всегда есть соблазн определить

```
1 vector<int> v1 = {1, 2, 3, 4};
2 vector<int> v2 = {10, 20, 30, 40};
3 assert(v1 + v2 == vector<int>{11, 22, 33, 44});
```

Например так можно было бы сделать, создав отдельную специализацию `vector` для типов, похожих на интегральные.

**Вопрос к студентам:** почему так делать нельзя?

Вместо этого в стандарт включен специальный класс `valarray`. У этого класса сложная судьба: будучи введён для математических вычислений, он, с развитием метапрограммирования, был практически вытеснен такими библиотеками как `Eigen`, основанными на шаблонах.

Тем не менее, в него было заложено несколько крайне интересных концепций, в том числе серьёзно упрощающих векторизацию операций над ним, и вполне достойных отдельного рассмотрения.

Во-первых, конечно, `valarray` ведёт себя именно как вектор при сложении.

```
1 valarray<int> v1 = {1, 2, 3, 4};
2 valarray<int> v2 = {10, 20, 30, 40};
3 assert(v1 + v2 == valarray<int>{11, 22, 33, 44});
```

И даже умножение у него поэлементное

```
1 valarray<int> v4 = v1 * v2 + v1 + v2; // { 62, 123, 305, 567 }
2 valarray<int> v4 = pow (v1, 2); // 4, 9, 25, 49
```

Но настоящая мощь `valarray` в том, что для него существуют векторные указатели (*slices*).

```
1 valarray<int> row(n);
2 slice red(0, n/3, 3);
3 row[red] = 255;
```

Этот код установит каждую третью ячейку `row` в значение 255. Векторный указатель `slice` имеет начало, конец и инкремент, он похож на запись цикла и действительно можно было бы записать (но слайс эффективней):

```
1 for (int i = 0; i != n; i += 3)
2     row[i] = 255;
```

Вторым важным контейнеро-подобным классом является жалкая попытка комитета отыграться за `vector<bool>`, а именно битовая маска `std::bitset`

На самом деле, конечно, `bitset` это альтернатива `array<bool>` то есть у него фиксированный размер, являющийся параметром контейнера. При этом он хранит данные более компактно (тоже суббайтно, как и `vector<bool>`).

```
1 // 24-bit number
2 bitset<24> s1 = 0x7ff000;
3 bitset<24> s2 = 0xff00;
4 s1[0] = 1; // either s1.set(0) or s1.set(0, 1)
5 auto s3 = s1 & s2; // s3 = 0xf000
```

Вообще хранение чего угодно ниже минимально адресуемого адресного пространства выглядит в трансляционной модели языка странно и, по идеи, должно приводить к проблемам. Но в данном случае речь о хорошо изученном механизме, относительно которого накоплен определённый опыт.

Конечно очень не хватает для битовых масок слайса, но слайса нет и для обычной битовой арифметики.

TODO: неудавшийся `array_view` и грядущий `span`

## 5.5 Ввод и вывод

*From a programmer's point of view,  
the user is a peripheral  
that types when you issue a read request*

— Peter Williams

Что-то про ввод и вывод уже было сказано в (2.7.1), здесь базовые вещи будут считаться уже понятными, но многое сознательно будет изложено повторно.

Ввод и вывод это самая старая часть нынешней стандартной библиотеки C++ и единственная её часть, которая действительно широко использовалась до стандартизации языка. Причём, эта подсистема действительно странная штука. Пусть необходимо просто распечатать целое число в 16-ричном формате. Как к этому подходят разные языки:

- **C** – `fprintf(stdout, "%x", n)`
- **Python** – `print("%x" % n)`
- **Java** – `System.out.println(String.format("%x", n))`
- **Perl** – `printf "%d\n", n`
- **C#** – `Console.WriteLine("{0:x}", n);`
- **Rust** – `println!("{}{:x}", n)`
- **Go** – `fmt.Printf("%x", n)`

А теперь внимание, как это выглядит на C++

`1 std::cout << std::hex << n;`

Причем многие языки из перечисленных выше, были, на самом деле, созданы гораздо позже C++. Это наводит на мысли о том, так ли удачна была разработанная в свое время для C++ система вывода, если все новые языки с криком разбегаются от неё, предпочитая копировать сишину с разными улучшениями? Но то, что застыло в истории языка, застыло там навсегда. Странная система стандартного ввода-вывода в C++ имеет место, она не всегда может быть заменена старыми добрыми сишными возможностями, ей часто (и обоснованно) пользуются и её лучше знать, чем не знать.

### 5.5.1 Проблемы с C-style IO

С точки зрения программиста на C++, C-style IO, хотя и доступен в языке, но имеет массу проблем. Вот только некоторые:

- Нерасширяемость. Например как определить новый форматный спецификатор?
- Неочевидность: выбор спецификатора определяется размером, который может не быть известен. Пример: `int64_t` требует препроцессора для крайне “читаемого” вывода в стиле: `printf("x = %" PRIu64 "d", x)`
- Небезопасность относительно типов: `printf("%s\n", 1)` не просто скомпилируется, а ещё и исполнится, приведя при этом к таким проблемам, которые вы поседеете искать и отлаживать.
- Небезопасность относительно количества аргументов.
- Нерасширяемость самого механизма файловых дескрипторов (ограниченных `FILE*`).

Чтобы решить все эти проблемы, язык C++ вводит несколько уровней абстракции: главным объектом связывающим воедино ввод и вывод становятся не файлы, а **потоки**, которые отдают техническую реализацию ввода и вывода своим буферным классам, а форматирование своим локалем с присущим им фасетами. Сами же они занимаются тем, что предоставляют ко всему этому сложный интерфейс, построенный во многом на переопределении операторов и прочих уникальных языковых возможностях.

При разговоре о переопределении операторов была высказана дельная мысль о том, что стоит избегать семантически безумных определений. Например плохая идея переопределять `operator *` для произведения матриц, так как оно не коммутативно и т.д. Обо всем этом на время чтения этого раздела стоит забыть: большего безумия, чем переопределить битовый сдвиг для ввода и вывода и представить себе невозможно. С этим нужно смириться.

Кроме того, потоки ввода/вывода (`streams`) не следует путать с потоками исполнения (`threads`), это очень разные вещи. Это настолько разные вещи, что я даже убрал слово “потоки” из заголовка раздела, чтобы не вводить в искушение во-первых и потому что оно уже есть в названии раздела про потоки исполнения во-вторых.

### 5.5.2 Иерархия классов для IO

Хорошим тоном в литературе считается сразу вдохновлять читателей иллюстрацией иерархии потоков ввода-вывода.

TODO: тут картинка

Несмотря на запутанность общей картины, глаз сразу выделяет основные паттерны. Есть две независимых иерархии для буферов с корнем в `basic_streambuf` и для потоков с корнем в `basic_ios` (который агрегирует `basic_streambuf`, поскольку каждый поток владеет своим буфером). Иерархия потоков отчетливо ветвится на ввод и вывод, а потом ромбовидно сходится на `basic_iostream`, от которого наследуют гибриды.

Вся эта иерархия это не столько иерархия классов, сколько иерархия шаблонов. Собственно классы, такие как `ostream` или `fstream` инстанцируют соответствующие шаблоны, определяя тип символов, который будет в них выводиться. Типы символов бывают разные, но об этом позже, а пока представим, что `char` хватает для всех.

### 5.5.3 Форматирование

Как ни странно, форматированный ввод и вывод оказывается гораздо проще неформатированного.

```
1 int n = 42;
2 cout << n << endl; // 42
3 cout << hex << n << endl; // 2A
```

Форматный модификатор меняет состояние потока, поэтому возвращается от шестнадцатиричного вывода к десятичному, нужно явно.

```
1 cout << hex << n << " " << dec << n << endl; // 2A 42
```

Аналогично ввод. Можно поиграть с простой программой, которая ждёт 16-ричное число и переводит его в десятичное.

```
1 int n;
2 cin >> std::hex >> n;
3 cout << std::dec << n << std::endl;
```

Например если ввести 2A и нажать ввод, программа выведет 42 и завершит работу.

**Вопрос к студентам:** а что будет если ввести нечто странное, вроде АААААААААААААААААААА?

Полный список стандартных манипуляторов всегда можно посмотреть в стандарте (вариант для слабаков – нагуглить в интернетах). В основном они разделяются на:

- Форматные модификаторы такие как `hex` и `dec` для целых или `fixed` и `scientific` для чисел с плавающей точкой.
- Специальные манипуляторы, такие как `endl` или `ends`
- Независимые флаги: `skipws`, `showpos`, `uppercase`, `boolalpha` и т.д. Обычно они все имеют отрицательные пары: `noskipws`, `noshowpos` и так далее

Например в приведенной выше программе, ввод “`2A`” будет засчитан как `42`, потому что пробелы не учитываются по умолчанию. Но это можно изменить:

```
1 cin >> std::hex >> std::noskipws >> n;
```

Теперь ввод лишних пробелов будет ошибкой.

- Параметризованные манипуляторы, такие как `setfill` или `setbase`, которые принимают дополнительные аргументы.

Вот так, например, можно вывести число в поле ширины 10, заполненное точками в пустых местах.

```
1 cout << std::setfill ('.') << std::setw (10) << 42;
```

В результате получится . . . . . 42. Точек восемь а не десять, потому что 10 это общая ширина поля для вывода.

Важно понимать, что все манипуляторы это просто функции, поэтому кроме использования перегруженных операторов сдвига, можно просто вызывать эти функции с потоком в качестве аргумента.

```
1 cout << hex; // operator << (cout, hex)
2           // calls hex(cout);
3 hex(cout); // call immediately
```

Обе записи равноправны, но первая бывает более удобна, когда в одной цепочке надо часто переключать манипуляторы.

Манипуляторы бывают очень простыми. Например манипулятор `endl` может быть устроен как-то так:

```
1 template <typename charT, typename traits>
2 std::basic_ostream<charT,traits>&
3 std::endl (std::basic_ostream<charT,traits>& strm)
4 {
5     strm.put(strm.widen'(\n));
6     strm.flush();
7     return strm;
8 }
```

Пока что тут непонятна функция `widen`, но интуитивно ясно, что она приводит символ конца строки к правильному формату конца строки в потоке, который может быть параметризован иным типом символов. Такие манипуляторы вполне можно писать самому. Но эта относительная простота обманчива: она сохраняется лишь пока нет необходимости работать с состоянием потока. Как ни странно, но разговор о написании собственного манипулятора, работающего с этим состоянием (как это делает, например `std::hex`, придётся отложить до рассмотрения локалей и фасетов (см. 5.6)).

И того, что операторы ввода и вывода это перегруженные операторы логического сдвига, следует, что их можно переопределять для любого класса

```
1 class MyClass
2 {
3     // something private
4 public:
5     // print have access to private data
6     void print (std::ostream& stream) const;
7 }
8
9 std::ostream& operator <<(std::ostream& stream, const MyClass&
10    rhs)
11 {
12     rhs.print (stream);
13     return stream;
14 }
```

Многие любят использовать для операторов вывода дружбу. Дело вкуса, но мне кажется, не стоит лишний раз давить кишки наружу, нарушая инкапсуляцию класса.

### 5.5.4 Неформатный ввод и вывод

Ввод без форматирования позволяет снова разделить ответственности: теперь потоки ввода оказываются нужны только для чтения (например посимвольного или построчного), а весь разбор программа берет на себя. Обычная ситуация в мире C++ это сочетания неформатированного ввода с форматированным выводом, потому что в форматированном вводе куда как сложнее обрабатывать ошибки.

Основные средства для посимвольного чтения из потока:

- `get` – считывает из потока один символ
- `peek` – подсматривает в потоке один символ
- `putback` – кладёт считанный символ обратно в поток. Стандартом гарантируется возможность положить обратно один только что считанный символ, но реализации иногда позволяют больше.

Можно поиграть со следующим кодом.

```
1 char c = cin.get();
2 if ( (c >= '0') && (c <= '9') )
3 {
4     int n;
5     cin.putback (c);
6     cin >> n;
7     // process number
8 }
9 else
10 {
11     string str;
12     cin.putback (c);
13     getline (cin, str); // not cin.getline (char *, int)
14     // process string
15 }
```

**Вопрос к студентам:** можно ли использовать `peek`, чтобы переписать его чуть лучше?

Особое внимание нужно обратить на `getline` из заголовочника `<string>`. Дело в том, что у потока ввода есть свой `getline` но он был

спроектирован раньше, чем `string` и не умеет динамически увеличивать буфер, а требует заранее заданного размера.

Особенности ввода иногда порождают сложные и трудно отлавливаемые проблемы.

```
1 cout << "Please enter a number: " << "\n";
2 cin >> num;
3 cout << "Your number is: " << num << "\n";
4 cout << "Please enter your name: \n";
5 getline (cin, mystr);
```

Эта программа кажется совершенно правильной... вот только она не считывает имя пользователя.

**Вопрос к студентам:** как вы думаете в чем дело?

Дело в том, что после ввода числа, в потоке ввода “залип” перевод строки. Для того, чтобы его оттуда убрать, используется метод `ignore`

```
1 cout << "Please enter a number: " << "\n";
2 cin >> num;
3 cout << "Your number is: " << num << "\n";
4 cout << "Please enter your name: \n";
5 cin.ignore();
6 getline (cin, mystr);
```

На самом деле этот метод имеет параметры: сколько символов пропригнорировать и на каком остановиться, но чаще всего он вызывается именно так для игнорирования одного символа (обычно возврата строки).

### 5.5.5 Состояния и обработка ошибок

Каждый поток ввода/вывода находится в том или ином состоянии после каждой операции. Обычно для потоков вывода это не слишком критично, так как сложно представить себе ошибку вывода (впрочем при выводе в файл или строку они бывают). Поэтому в 99 процентах случаев проверка состояний используется для потоков ввода.

Состояние задаётся тремя битами: `eofbit`, `failbit` и `badbit`. Если ни один из этих битов не установлен, состояние считается хорошим (иногда используется термин `goodbit`, но автору он не нравится, так как нет такого бита). Различия между тремя плохими состояниями довольно просты:

- `std::ios_base::eofbit` – считан конец файла (EOF можно проэмулировать с клавиатуры, в unix-системах это Ctrl-D)
- `std::ios_base::failbit` – восстановимая ошибка. Например ошибка форматирования, когда ожидается число, а считана строка
- `std::ios_base::badbit` – серьёзная ошибка: поток испорчен или информация потеряна. Например позиционирование потока раньше начала файла, etc.

Обычно `eofbit` и `failbit` случаются вместе, так как почти ни один файл в нашем мире не завершается символом конца файла, вместо этого конец файла детектируется при неудачной попытке считать после конца файла.

Также в `std::ios_base` поддерживается функция `rdstate` позволяющая считать флаги и функция `clear`, позволяющая их сбросить. Особое место занимает функция `fail`, которая возвращает `true`, если установлен `failbit` или `badbit`.

Для потоков также переопределены два оператора: неявное преобразование `operator bool()` соответствует `!fail()` и оператор отрицания (о нет, опять) `operator !()`, соответствующий `fail()`.

Можно попробовать программу:

```
1 int n;
2
3 while (cin >> n)
4 {
5     cout << n << endl;
6     cin.ignore(); // eating Enter hit
7 }
```

**Вопрос к студентам:** что она выведет если вы введете 1.1?

С этой проверкой с неявным преобразованием связана глупая ошибка. Люди пишут:

```
1 if (!cin >> n)
2 {
3     // process errors
4 }
```

Что при этом случается: сначала вызывается `operator !()`, который возвращает текущее состояние потока, а потом оно приводится к целому

и сдвигается вправо на n. Наверное это не то, чего вы хотели. То, что вы хотели, достигается расстановкой скобок.

```
1 if (!(cin >> n))
2 {
3     // process errors
4 }
```

Но на самом деле, то, чего вы хотели достигается тем, что вы прекращаете вынендриваться и пишете прозрачный и читаемый код:

```
1 cin >> x;
2 if (cin.fail()) {
3     // process errors
4 }
```

Интересно, что из-за почтенного возраста библиотеки, вся обработка ошибок производится без использования исключений и это идёт скорее на пользу: ошибка пользовательского ввода это почти никогда не исключительная ситуация. Механизмы завести ошибки потоков на исключения есть, но я о них рассказывать не буду, чтобы не навредить.

### 5.5.6 Работа с файлами

Настоящая мощь потоков раскрывается через прозрачную работу с файлами. Главное преимущество это RAII (см. 3.2.1) выражющееся в том, что файлы живут под потоками, открываются и закрываются по необходимости. Всё, что нужно в простейшем случае, это имя файла и иногда режим в котором его открыть.

Первый пример из [10]: записать текущий charset в файл

```
1 {
2     ofstream file(filename);
3
4     if (!file)
5     {
6         // process errors
7     }
8
9     // write character set
10    for (int i = 32; i < 256; ++i)
11    {
```

```

12     file << "value: " << setw(3) << i << " "
13         << "char: " << static_cast<char>(i) << endl;
14     }
15
16 } // scope ends

```

Файл будет автоматически закрыт в конце своей области видимости.

Второй пример: выдать содержимое файла на экран:

```

1 ifstream file(filename);
2
3 // check for errors like above
4
5 char c;
6 while (file.get(c))
7     cout.put(c);

```

Это сразу вызывает вопрос: а что если файл нужен мне дольше, чем его область видимости? Раньше в C++98 это приводило к уродливым хакам, вроде создания потока в динамической памяти. Но в C++11 для файлов определен move ctor и следующий код становится легальным:

```

1 std::ofstream
2 openFile (const std::string& filename)
3 {
4     std::ofstream file(filename);
5     file << "hello, ";
6     return file;
7 }
8
9 std::ofstream file;
10 file = openFile("xyz.tmp");
11 file << "world" << std::endl;

```

Возможен и обратный вопрос: можно ли один поток использовать для нескольких файлов? И снова ответ положительный: для этого используются функции `open` и `close`, позволяющие закрыть нижележащий файл и заново его открыть.

Файлы могут быть открыты с тонким управлением режимами. Например если записанный внутри функции файл был там закрыт, его можно открыть на дозапись (append) скомбинировав флаги вручную.

```
1 ofstream file("a.tmp", std::ios_base::out|std::ios_base::app);
```

Полный список флагов несложно посмотреть в стандарте. Гораздо полезнее знать их соответствие распространённым флагам из мира С.

| Тип                    | Файл                           | Поток                            |
|------------------------|--------------------------------|----------------------------------|
| Чтение                 | " <b>r</b> "                   | <b>in</b>                        |
| Перезапись             | " <b>w</b> "                   | <b>out</b> или <b>out trunc</b>  |
| Дозапись               | " <b>a</b> " или " <b>a+</b> " | <b>app</b> или <b>in out app</b> |
| Чтение/запись с начала | " <b>r++</b> "                 | <b>in out</b>                    |
| Чтение/перезапись      | " <b>w++</b> "                 | <b>in out trunc</b>              |

Файловые потоки вывода при записи можно позиционировать

```

1 outfile << "This is an apple";
2 long pos = outfile.tellp();
3 outfile.seekp(pos - 7);
4 outfile << " sam";
```

**Вопрос к студентам:** что окажется в файле?

Приведенный выше код обычно работает, но после 2011 года хорошим тоном считается не закладываться на то, что результат `tellp` можно преобразовать к `long` и обратно. Более правильным будет написать:

```

1 auto pos = outfile.tellp();
2 outfile.seekp (pos - static_cast<decltype(pos)>(7));
```

Это чуть дольше, но это уберегает от неприятностей при поддержке кода.

Ещё лучше сформулировать относительный поиск как относительный. Если всё что нужно это перебазировать на семь позиций от текущей позиции, то сработает следующий подход.

```

1 outfile << "This is an apple";
2 outfile.seekp (-7, std::ios::cur);
3 outfile << " sam";
```

Он, пожалуй, и является в этом случае оптимальным. Разумеется, наравне с `cur` доступна относительная привязка к `beg` и `end`.

Не все потоки поддерживают позиционирование, например его по понятным причинам не поддерживают `cin` и `cout`.

### 5.5.7 Бинарный ввод и вывод

Файлы бывают не только текстовые.

TODO: тут про бинарный ввод и вывод.

### 5.5.8 Работа со строками

Строка является удивительным объектом для ввода и вывода – по сути она не связана ни с какой аппаратурой, а является собой просто буфер в памяти с приятным интерфейсом доступа и форматирования.

Строковые потоки могут быть созданы как независимо, так и из строк. Пример должен быть уже понятен.

```
1 std::ostringstream fst;
2 int n;
3 float f;
4
5 fst << 42.2442;
6 std::string s1 = fst.str();
7 std::istringstream iss(s1);
8 iss >> n >> f;
9 std::string s2("value: ");
10 std::ostringstream snd (s2, std::ios::out|std::ios::ate);
11 snd << std::hex << n << " " << f;
12 std::cout << snd.str() << std::endl;
```

Сначала в пустой и чистый строковый поток выводится значение числа с плавающей точкой. Потом его результат извлекается в строку. Далее из этой строки создаётся поток ввода и из него извлекается целое и число с плавающей точкой. Далее создаётся строка с текстом, потом из неё поток и в этот поток извлеченное целое выводится в шестнадцатеричном виде. И наконец, всё это оказывается на экране.

**Вопрос к студентам:** что окажется на экране?

Строковые потоки даже не обязаны быть именованными (код ниже требует C++11).

```
1 void parseName(string name)
2 {
3     string s1, s2, s3;
4     istringstream(name) >> s1 >> s2 >> s3;
```

5        . . . .

Разумеется, как и файловые потоки они поддерживают и семантику перемещения и позиционирование.

Ранее (см. 5.1.3) была рассмотрена гипотетическая функция `combine`.

TODO: тут комбинатор на вариабельных шаблонах

### 5.5.9 Буферизация

Как и было сказано (см. 2.7.1), с каждым потоком связан буфер (возможно нулевого размера). При этом буферизация бывает построчная полная или никакая. Средства C++ тоже позволяют управлять буферизацией.

Уже несколько раз был упомянут прямой вывод содержимого любого потока на экран

```
1 char c;
2 while (file.get(c))
3     cout.put(c);
```

Если ещё раз вернуться к коду выше, то он очевидно неоптимален. Оптимальный метод выглядит даже проще:

```
1 cout << file.rdbuf();
```

Странная функция `rdbuf` имеет два предназначения: без аргументов она возвращает указатель на буфер потока, с аргументом она устанавливает буфер. Например дебуферизовать стандартный ввод можно очень легко.

```
1 cout.rdbuf(nullptr);
```

Но обычная задача это как раз наоборот: задать буфер побольше для файлового потока, чтобы минимизировать количество чтений с диска и записей на диск.

Разумеется, для буферизованных потоков нужно не забывать сбрасывать их содержимое из буфера на устройство и делать это приходится руками. Тем не менее, есть некоторые иные механизмы для распространенных случаев. Например, кажется, что этот код не должен ничего печатать на экран до ввода:

```
1 std::cout << "Please enter x: ";
2 std::cin >> x;
```

Но он печатает. Это происходит по двум причинам:

1. стандартный ввод в C++ связан (tied) со стандартным выводом
2. стандартный вывод cout в C++ синхронизирован (имеет один буфер) со старым добрым stdout

Если развязать обе эти связи, то поведение кода становится ожидаемым:

```
1 std::cin.tie(NULL);
2 std::cout.sync_with_stdio(false);
3 std::cout << "Please enter x: ";
4 std::cin >> x;
```

Функция `tie` связывает потоки отношением “случается до”. То есть заставляет все побочные эффекты одного потока случится до начала работы с другим. Но можно связать потоки ещё крепче, заставив их физически использовать один буфер:

```
1 ostream hexout(std::cout.rdbuf());
2 hexout.setf(std::ios::hex, std::ios::basefield);
3 hexout.setf(std::ios::showbase);
4
5 std::cout << 42 << " ";
6 hexout << 42 << std::endl;
```

На экране будет: “42 0x2a”. Это вывод в один и тот же стандартный вывод, только теперь он по-разному форматирован.

**Домашняя наработка:** почему вывод указателя на буфер потока ввода выводит в поток вывода не указатель, а всё, что вводится в поток ввода?

## 5.6 Локализация

Не все буквы можно поместить в диапазон однобайтных символов. Пока речь идёт только об английском языке, всё неплохо, но русские буквы, европейская диакритика, и даже иероглифическое письмо также имеют свое место в мире C++.

В доисторические времена считалось, что английских символов хватит на всех. Из этих времён до нас дожил стандарт ANSI, предполагающий 7-битные кодировки. Его расширение ASCII предлагало 8 бит, первые 128 символов совпадали с ANSI, а оставшиеся 128 определялись **кодовой страницей**. В качестве примера всем известны CP-1251 и CP-866.

Тогда же возникли многословные (multibyte) строки и форматы. Например если зафиксировать байт 0x90 для специального случая (следующий символ означает нечто иное), то можно получить 511 символов вместо 256. Люди фиксировали целые диапазоны и как-то крутились. Проблема в том, что в такой кодировке очень сложно работать таким функциям как strcmp или strlen.

Система Unicode предлагает отличие между системой кодировки (Universal Character Set, UCS) и форматом преобразования (Universal Translation Format, UTF).

Основные системы кодировки это UCS-2, кодирующая символы по 16 бит и UCS-4, кодирующая по 32 бита. С последней возможны широкие (wide) кодировки, когда каждый символ занимает по 4 байта и этого хватает на всех. Но до сих пор самые частые символы это старые добрые символы из ANSI, которые можно кодировать одним байтом.

Поэтому трансляции поддерживаются следующие:

1. UTF-8 (от 1 до 6 байт на символ) в ней U+0410 это 0xD0, 0x90, зато U+0041 это 0x41
2. UTF-16 (покрывает UCS-2) в UTF16-LE U+0410 это 0x10, 0x04 но и U+0041 это 0x41, 0x00
3. UTF-32 (покрывает UCS-4)

Цель этого раздела показать как эти возможности ложатся на язык и к чему это приводит.

В английском языке слово “интернационализация” (internationalization) весьма длинное с точки зрения носителей языка, поэтому оно часто сокращается до “i18n”. Также в литературе используется “l10n” для “локализации” (localization).

### 5.6.1 Символы и характеристики символов

Стандарт определяет четыре основных типа для представления символов:

- **char** – тип минимального размера. `sizeof(char)== 1`
- **char16\_t** – символьный тип для UCS-2
- **char32\_t** – символьный тип для UCS-4
- **wchar\_t** – наиболее широкий символьный тип из всех, присутствующих в системе

Обычно символьный тип является главным параметром шаблона для таких классов, как строки, потоки и им подобных. Например общий шаблон потока

```
1 template <typename charT, typename traits = char_traits<charT>>
2 class basic_istream;
```

Далее определяется для разных типов символов в разные синонимы.

```
1 typedef basic_istream<char> istream;
2 typedef basic_istream<wchar_t> wistream;
```

Гораздо более интересен второй аргумент, который идёт по умолчанию: характеристики символов. Одни и те же по ширине символы могут подчиняться разным правилам сравнения, копирования, иметь разный размерный тип и даже разное представление EOF (подробнее см. 5.1.6).

По умолчанию характеристики определены для каждого из стандартно поддержанных символов, но никто не ограничивает в том, чтобы определить и передать свои.

### 5.6.2 Локали

Локали инкапсулируют национальные и культурные конвенции. Самый простой способ идентификации локали это идентификация текстовой строкой в формате “lang[\_area[.code]][@modifier]”, например валидная локаль для POSIX это: “de\_DE.ISO-8859-1”.

Увы этот метод не слишком надежен и не слишком стандартен. Та же самая локаль под Windows будет называться уже “deu\_DEU.ISO-8859-1”.

Единственная локаль, которая обязана присутствовать по имени это “C”. Также пустое имя используется для локали по умолчанию

В языке С локали устанавливаются только строками.

```
1 setlocale (LC_ALL, "")
```

Здесь LC\_ALL это набор характеристик, которые устанавливаются: сборная константа, которая включает в себя флаги LC\_COLLATE, LC\_CTYPE, LC\_MONETARY, LC\_NUMERIC, LC\_TIME.

После установки локал и в языке С и в C++ можно использовать функции стандартной библиотеки, такие как `strcoll` для сравнения строк с учетом локалей, вывода дат и сумм и так далее.

```
1 setlocale(LC_COLLATE, "cs_CZ.iso88592");
2
3 const char* s1 = "hrnec";
4 const char* s2 = "chrt";
5
6 cout << "In the Czech locale: ";
7 if(strcoll(s1, s2) < 0)
8     cout << s1 << " before " << s2 << '\n';
9 else
10    cout << s2 << " before " << s1 << '\n';
11
12 cout << "In lexicographical comparison: ";
13 if(strcmp(s1, s2) < 0)
14     cout << s1 << " before " << s2 << '\n';
15 else
16    cout << s2 << " before " << s1 << '\n';
```

На самом деле в C++ назначение глобальной локали лучше делать не через унаследованные функции, а явно

```
1 locale::global(locale(""))
```

Также объект, конструируемый в прошлом примере позволяет получить текущую локаль по имени:

```
1 cout << locale("").name() << endl;
```

Но одна глобальная локаль на всех это как-то не очень. C++ предоставляет гораздо больше гибкости, в частности локали могут быть назначены (imbued) отдельным потокам, в том числе разным – разные.

```
1 cin.imbue(locale(""));
2 // POSIX only, for Windows use "rus_RUS"
3 cout.imbue(locale("ru_RU"));
4 double value;
5 if (cin >> value) cout << value << endl;
```

Теперь ввод в локали по умолчанию (например “47.11”) будет сопровождаться выводом в немецкой локали “47,11”.

### 5.6.3 Фасеты

Назначаемость локалей это уже неплохо, но пока что назначались те же локали, идентифицируемые строками. Что насчет изменения и построения собственных? В языке C++ предусмотрена эта возможность.

Дело в том, что локаль это ни что иное как просто контейнер фасетов. Фасет это наследник класса `std::locale::facet`, который не так уж и сложен. По сути он только запрещает конструктор копирования. Так что можно сказать, что фасет это просто нечто не копируемое.

TODO: завершить про фасеты

## 5.7 Итераторы

Этот раздел будет посвящён итераторам. Базовые представления об итераторах уже были даны при разговоре о контейнерах. Систематизируя то, что уже известно, можно сказать:

- Итератор это объект, описывающий позицию в контейнере
- Базовые итераторы получаются встроенными функциями `begin()` и `end()` которые должен предоставлять любой совместимый контейнер
- Далее он может быть разыменован, продвинут, сравнён на неравенство

На самом деле итераторы это довольно сложная концепция, со своими ловушками, достоинствами и недостатками. Но главное – эта концепция незаменима для обобщённого программирования и именно поэтому так плотно прибита гвоздями к стандартной библиотеке.

### 5.7.1 От указателей к итераторам

Допустим, есть массив и функция, которая обходит этот массив и что-то на нем выполняет:

```
1 template <typename F>
2 size_t traverse (vector<int> &v, F func) {
3     size_t nelts = v.size();
4     for (size_t i = 0; i != nelts; ++i)
5         if (!func(v[i]))
6             return i;
7     return nelts;
8 }
```

Пока что кажется, что всё хорошо и это именно тот код, который написало бы большинство людей без опыта в языке, если бы им встретилась такая задача. Но нет. На самом деле этот код фундаментально плох. И то, насколько он плох показывает попытка его обобщить. Итак, пусть теперь конкретный тип контейнера тоже задаётся шаблонным параметром. Простое переписывание даёт:

```

1 template <typename Cont, typename F>
2 size_t traverse (Cont &cont, F func) {
3     size_t nelts = cont.size();
4     for (size_t i = 0; i != nelts; ++i)
5         if (!func(cont[i]))
6             return i;
7     return nelts;
8 }
```

Теперь очевиден недостаток: этот код требует от типа `Cont` слишком много: как минимум контейнер должен предоставлять произвольный доступ. А что если контейнер `list` или `forward_list`?

Чтобы всё работало, необходимо, как легко догадаться из названия раздела, использовать итераторы.

```

1 template <typename Cont, typename F>
2 size_t traverse (Cont &cont, F func) {
3     size_t elts = 0
4     for (auto it = cont.begin(); it != cont.end(); ++it, ++elts)
5         if (!func(*it))
6             break;
7     return elts;
8 }
```

В этом коде тип `Cont` должен удовлетворять следующим условиям: он содержит функцию `begin()` для возврата итератора на своё начало и `end()` для итератора на свой конец, итератор может быть сравнён на равенство и неравенство, разыменован и инкрементирован. Это почти любой контейнер STL.

Говорят, что `begin()` и `end()` задают **диапазон**. Сама идиома “итерации в диапазоне, задаваемом контейнером”, настолько распространена, что получило специальную синтаксическую поддержку в так называемом обходе диапазона (range-based for).

```

1 size_t elts = 0
2 for (auto elt : cont)
3     if (!(++elts, func(elt)))
4         break;
5 return elts;
```

По стандарту это работает следующим образом:

```
1 for (range_declaration : range_expression)
```

```
2     loop_statement;
```

Эквивалентно следующему (начиная с 2017 года, до этого было семантически то же, но с проблемами).

```
1 auto && __range = range_expression;
2 auto __begin = std::begin(__range);
3 auto __end = std::end(__range);
4 for ( ; __begin != __end; ++__begin) {
5     range_declaration = *__begin;
6     loop_statement
7 }
```

Собственно определение того, чем является range-based for менялось дважды – в 2014 и 2017 годах и каждый раз семантически оставалось неизменным.

Обычно используется аннотированный и уточненный вид `auto` чтобы не попадать на лишние копирования

```
1 for (const auto &n : v)
2     cout << n;
```

Обход диапазона требует от итераторов контейнера совсем немногого: инкремент, разыменование, сравнение на неравенство. Поэтому даже очень простые пользовательские контейнеры позволяют такой обход.

```
1 template <int S> class MyArray {
2     int arr[S];
3 public:
4     template <typename ... Ts>
5     MyArray (Ts ... ints) : arr {ints ...} {}
6 public:
7     int *begin() { return arr; }
8     int *end() { return arr + S; }
9 };
```

Здесь в роли итераторов выступают обычные указатели и это ничего не меняет.

```
1 MyArray<6> marr = {1, 2, 3, 4, 5, 6};
2 // range-based traverse works!
3 traverse (marr, [](int& n) { cout << n << " "; return true; });
```

Кроме того, очень часто можно увидеть **конструирование из диапазона**. Это часто встречающаяся идиома и её истоки стоит рассмотреть подробней. Допустим, при написании класса `vector`, появилась необходимость дать пользователю возможность сконструировать его из самых разных классов:

```
1 vector (const list<T> &);
2 vector (const set<T> &);
3 vector (const T * pod_array, int n); /* ! */
```

Следует обратить внимание, что в итоге получился разный интерфейс для конструирования из списка, который знает свою длину и из POD-массива, который свою длину не знает. Кроме того, конструкторов получилось слишком много. Пара итераторов здесь позволяет естественным образом обобщить идею “диапазона” в одном конструкторе:

```
1 template <class InputIterator>
2 vector (InputIterator begin, InputIterator end)
```

Такой конструктор в классе `vector` действительно есть. Теперь любой список и множество могут сообщить вектору свои начало и конец, а POD-массив – использовать для этого обычные указатели.

```
1 vector<int> v(marr.begin(), marr.end());
```

Поддержка диапазонных конструкторов – хороший тон в контейнере, так как позволяет быстро перегонять в него любой другой.

**Вопрос к студентам:** напишите диапазонный конструктор для `MyArray`

Внезапная проблема с диапазонными конструкторами возникает в таких классах как `vector`, где могут встречаться другие конструкторы с двумя аргументами. Рассмотрим простой вроде-как-вектор

```
1 template <typename T> class MyVector {
2     T *arr_;
3     size_t size_;
4 public:
5     MyVector (size_t nelts, T value);
6     template <typename Iter> MyVector (Iter fst, Iter lst);
7     // .... everything else ....
8 };
```

Попытка использовать его первый конструктор будет неудачна и будет вызван второй (скорее всего с ошибкой, так как целое число нельзя разыменовать, но если ошибки там и не будет, это будет ещё хуже).

```
1 MyVector<int> mvec (2, 2); // FAIL
```

Обычно для решения таких проблем используется техника SFINAE (см. 4.8.3). Простейший способ это проверить итерабельность аргументов на этапе шаблонной подстановки.

```
1 MyVector (size_t nelts, T value);
2 template <typename Iter,
3           typename = decltype(*declval<Iter&>(),
4                                ++declval<Iter&>(),
5                                void())> >
6 MyVector (Iter fst, Iter lst);
```

Обычно используется более систематичное SFINAE, выше приведён крайне колхозный пример.

Ещё одна серьёзная проблема с диапазонами подстерегает в неожиданном месте. Пусть задан алгоритм сортировки (такой действительно есть), берущий на вход два итератора. Как использовать его, чтобы отсортировать все элементы вектора кроме первого элемента?

```
1 template <typename Iter> sort (Iter fst, Iter lst);
2 vector<int> v = {10, 2, 5, 7, 3, 9};
3 sort (???, ???); // v == {10, 2, 3, 5, 7, 9};
```

Первый вариант решения, который приходит в голову, основан на инкременте.

```
1 sort (++v.begin(), v.end()); // FAIL
```

Он хорош всем, кроме одного: он не работает. Причём (и это обидно) не работает он не по каким-то эзотерическим причинам, а вполне банально: результат вызова функции `begin` это `rvalue`, а к `rvalue` не применим преинкремент. Правильное решение будет использовать вспомогательную функцию `next`

```
1 sort (std::next(v.begin()), v.end()); // OK
```

Основные вспомогательные функции для работы с итераторами это:

- `distance (fst, lst)` – расстояние между элементами в порядке итерирования
- `advance (it, n)` – продвинуться от текущего элемента на `n` позиций.

- `next (it)` – следующий элемент в порядке итерирования, как `advance (1)`
- `prev (it)` – предыдущий элемент в порядке итерирования, как `advance(-1)`

Все они работают с той сложностью, с которой получится, тут нет никаких гарантий. То есть `distance` для `list` это  $O(N)$  и т.д.

**Вопрос к студентам:** как вы предпочтёте мерить расстояние: функцией `distance` или прямой разностью итераторов?

### 5.7.2 Характеристики итераторов

Говорят, что итераторы это обобщение указателей. Это слово означает, что по возможностям указатели являются расширением итераторов. И действительно, итераторы по большей части ограничивают доступ к контейнеру. Например прямые итераторы (я буду иногда позволять себе англицизм форвард-итераторы, благо он ужеочно прижился в языке) дают возможность только инкрементировать себя, но не декрементировать (в то время как указатели всегда имеют обе этих возможности). А двунаправленные итераторы “отдают” декремент, но не позволяют использовать обращение через квадратные скобки.

По ограничениям доступа к элементам контейнера, итераторы делятся на:

- `output` – итераторы вывода, служат для вывода результатов в какое-то такое место, где эти результаты пропадают и более недоступны. Канонический пример – итератор для `ostream` (см. 5.7.3)
- `input` – итераторы ввода, то же что и выше, но для ввода. Канонический пример – итератор для `istream`
- `forward` – итераторы прямого доступа позволяют проходить последовательность только в одном направлении, но сколько угодно раз. Канонический пример – итератор для `forward_list`
- `bidirectional` – двунаправленные итераторы – то же, что и выше, но в обоих направлениях. Канонический пример – итератор для `list`

- random access – итераторы прямого доступа. Почти указатели (часто это указатели и есть). Канонический пример – итератор для `vector`

Можно ввести некоторые свойства перенумеровав их для наглядности:

1. Поддержка создания по умолчанию, копирующего конструктора и присваивания, а также деструктора

```
1 X a;  
2 X b(a);  
3 b = a;
```

2. Возможность разыменования как rvalue и доступа к полям по указателю

```
1 x = *a;  
2 y = a->t;
```

3. Возможность разыменования как lvalue

```
1 *a = t;
```

4. Поддержка инкремента и постинкремента

```
1 ++a;  
2 a++;
```

5. Сравнимость на равенство/неравенство

```
1 a == b  
2 a != b
```

6. Поддержка декремента и постдекремента

```
1 --a;  
2 a--;
```

Тогда сопоставляя наборы свойств, имеем:

- output – (1, 3, 4)

- input – (1, 2, 4, 5)
- forward – тоже (1, 2, 4, 5) но при этом подытераторный тип должен поддерживать присваивание
- bidirectional – (1, 2, 4, 5, 6)
- random access – (1, 2, 4, 5, 6) а также поддержка `operator []`, сложения с константой, вычитания константы и сравнения на больше и меньше.

Таким образом требование к чему-то “быть forward итератором” это аббревиатура к набору условий на этот тип. Для того, чтобы на этапе компиляции различать категории итераторов и даже делать перегрузки функций по этим категориям, служат **характеристики итераторов**.

Полный перечень характеристик совпадает с перечнем категорий и каждая характеристика это отдельный тип (ничего слишком интересного в себе не содержащий).

```
1 class input_iterator_tag;
2 class output_iterator_tag;
3 class forward_iterator_tag: public input_iterator_tag;
4 class bidirectional_iterator_tag: public forward_iterator_tag;
5 class random_access_iterator_tag: public
    bidirectional_iterator_tag;
```

Умелое использования характеристик позволяет создавать код, который по разному ведет себя в зависимости от возможностей итераторов без знания конкретных типов контейнеров. Для доступа к характеристике используется класс `iterator_traits`, параметризованный типом итератора. Применение станет ясно из следующего примера.

```
1 ostream&
2 operator << (ostream& out, random_access_iterator_tag) {
3     out << "random access"; return out;
4 }
5
6 ostream&
7 operator << (ostream& out, bidirectional_iterator_tag) {
8     out << "bidirectional"; return out;
9 }
10
11 // .... and so on for all tags ...
```

```
12
13 template <typename Iter> void
14 print_iterator_type (Iter it) {
15     cout << typename iterator_traits<Iter>::iterator_category{}
16     << endl;
}
```

Теперь можно поставить ряд экспериментов над распространёнными последовательными контейнерами:

```
1 print_iterator_type (forward_list<int>{}.begin());
2 print_iterator_type (list<int>{}.begin());
3 print_iterator_type (vector<int>{}.begin());
```

На экран будет выведено:

```
forward
bidirectional
random access
```

**Вопрос к студентам:** Можете ли вы не ставя эксперимент догадаться, что выведет на экран `print_iterator_type` для `deque<int>`?

### 5.7.3 Итераторы потоков ввода-вывода

Продолжая эксперименты, можно заметить, что в стандартных контейнерах нигде не встречаются `input` и `output` итераторы. И действительно они слишком ограничивающи для контейнеров и поэтому традиционно оставлены для использования с потоками ввода-вывода.

Это использование имеет ряд своих особенностей. Например не совсем ясно что такое итератор “конца потока”, если поток по определению не имеет информации о том, когда он закончит ввод. Здесь разработчиками языка принят ряд забавных решений, которые неплохо бы знать. Сначала, как обычно, пример.

```
1 istream_iterator<string> beg(cin), end;
2 vector<string> vec (beg, end);
3 copy (vec.begin(), vec.end(),
4        ostream_iterator<string>(cout, "\n"));
```

Эта простая программа считывает пользовательский ввод по стандартным разделителям (см. 5.5.3) в вектор, а потом выводит вектор, побитый переводами строки, на экран.

Использованные здесь типы `istream_iterator` и `ostream_iterator` также могут быть исследованы через механизм `print_iterator_type` (см. 5.7.2).

```
1 print_iterator_type (istream_iterator<int>());
2 print_iterator_type (ostream_iterator<int>(cout));
```

Разумеется результатами будут строчки `input` и `output`. В примере также видно, что итератор конца потока ввода это сингулярный (от англ. `single` – одиночный, не привязанный ни к какому потоку) `istream_iterator`. Это верно также для случаев, когда поток ввода привязывается к контейнеру, например в случае ввода из строки.

```
1 istringstream str("0.1 0.2 0.3 0.4");
2 copy (istream_iterator<double>(str),
3        istream_iterator<double>(),
4        ostream_iterator<double>(std::cout, " "));
```

В разделе (5.7.1) была приведена SFINAE-конструкция для того, чтобы решить проблему конструкторов у вектора. Там она выглядела как-то так (ниже только основная идея):

```
1 template <typename Iter,
2           typename = decltype(*declval<Iter&>(),
3                                ++declval<Iter&>(),
4                                void())> >
5 MyVector (Iter fst, Iter lst);
```

Можно подумать как использовать категории итераторов, чтобы сделать это решение лучше. В частности, указать, что конструктор может быть инстанцирован только для `input`-итераторов и производных от них.

```
1 template <typename Iter, typename = enable_if_t<
2           !is_integral<Iter>::value &&
3           is_base_of <
4             input_iterator_tag,
5             iterator_traits<Iter>::iterator_category
6           >::value
7         > >
8 MyVector (Iter fst, Iter lst);
```

Теперь решение гораздо систематичней, потому что вместо ad-hoc механизмов использует `enable_if` и стандартные функторы. К тому же оно куда читабельней (общий смех).

#### 5.7.4 Константные и обратные итераторы

По cv-квалификации и направлению доступа различают:

- `iterator`
- `const_iterator`
- `reverse_iterator`
- `const_reverse_iterator`

Каждый совместимый контейнер должен определять как минимум два, а лучше все четыре зависимых типа для итераторов:

```
1 for(vector<int>::iterator i = randomData.begin();
2     i != randomData.end();
3     ++i)
4 {
5     /* modify *i */
6 }
7
8 for(vector<int>::const_iterator i = randomData.cbegin();
9     i != randomData.cend();
10    ++i)
11 {
12     /* *i is readonly */;
13 }
```

Использованная выше функция `cbegin` принадлежит одной из четырёх пар функций.

- `begin` и `end`
- `cbegin` и `cend`
- `rbegin` и `rend`

- `crbegin` и `crend`

Важно следить за парностью таких функций. Попытка пробежать от `v.rbegin()` до `v.cend()` ничем хорошим не закончится.

Обратные итераторы важны также, потому, что пара (`end`, `begin`) не образует диапазона. Поэтому для того чтобы получить вектор в обратном порядке, вот такой код будет неверен:

```
1 vector<int> vecf = {1, 2, 3, 4, 5, 6};
2 vector<int> vecb (vecf.end(), vecf.begin()); // FAIL
```

Вместо этого, следует использовать

```
1 vector<int> vecb (vecf.rbegin(), vecf.rend());
```

Возвращаясь к указателям, можно изобразить простую диаграмму преобразований для указателей исходя из того, что там есть `const_cast` (см. рис. 5.20)

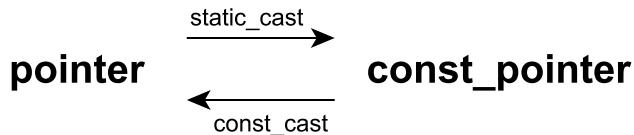


Рис. 5.20: Диаграмма преобразований для указателей

Для итераторов всё не так просто. Итераторы инвариантны и типы `iterator` и `const_iterator` не имеют друг с другом ничего общего (про ковариантность и инвариантность в приложении к контейнерам, можно прочитать в 5.4.3). Тем не менее, какие-то средства преобразования стандарт, разумеется, предусматривает. Первым их свёл в одну диаграмму Скотт Майерс [6] и диаграмма Майерса с учётом современных реалий выглядит как показано на (рис. 5.21)

Здесь стрелки отмечены цифрами:

- (1-4) Обращение итератора

```
1 auto rit = make_reverse_iterator(it);
2 auto it = rit.base();
```

- (5, 6) Добавление константности

```
1 Cont::const_iterator cit = it;
2 Cont::const_reverse_iterator crit = rit;
```

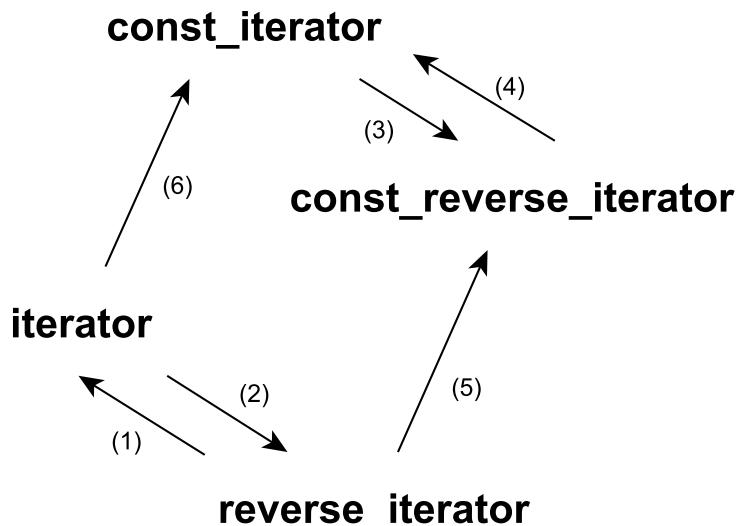


Рис. 5.21: Диаграмма преобразований для итераторов

В случаях 5 и 6, нужно иметь контейнер, чтобы выполнить приведение. Из диаграммы видна самая главная проблема: из константного итератора, как с Дону, выдачи нет. Майерс [6] для аналога `const_cast` предлагает использовать `advance`

```

1 Iter i(cont.begin());
2 advance(i, distance<decltype(ci)>(i, ci));
  
```

**Вопрос к студентам:** зачем явно указан шаблонный параметр для `distance`?

Основная проблема с этим подходом: время  $O(N)$  для “неудачных” контейнеров, таких, как списки. Решение для C++11 известно как “Hinant trick” и выглядит забавно.

```

1 template <typename Container, typename ConstIterator>
2 typename Container::iterator
3 remove_constness(Container& c, ConstIterator it) {
4     return c.erase(it, it);
5 }
  
```

Переход от обратного итерирования к прямому гораздо проще, особенно если понимать как работает функция `base`.

```

1 vector<int> v {1, 2, 3, 4, 5, 6, 7};
2 auto ri = v.rbegin() + 4;
3 auto it = ri.base();
  
```

```
4 cout << *ri << " " << *it << endl;
```

**Вопрос к студентам:** угадайте, что на экране?

Чтобы понять, что же на экране, полезно посмотреть на то как выглядит контейнер в прямом и в обратном порядке. Пояснение процесса см. на (рис. 5.22).

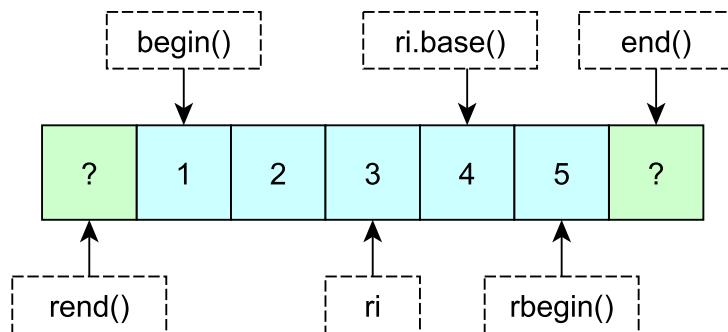


Рис. 5.22: Прямое и обратное итерирование

Можно думать об итераторе как о стрелочке, которая находится до (прямой итератор) или после (обратный итератор) элемента на который указывает и, при обращении, оборачивается на месте.

### 5.7.5 Адаптеры итераторов

Итераторы так же как и контейнеры можно **адаптировать** и особенно хорошо это видно на примере адаптации по направлению.

Выше уже рассматривался range-based for (см. 5.7.1).

```
1 for (auto elt : vec)
```

Такая запись обходит контейнер vec в прямом порядке.

Можно ли написать адаптер, чтобы обойти тот же контейнер в обратном порядке?

```
1 for (auto elt : reverse_cont(vec))
```

Оказывается да, можно. И, мало того, этот адаптер настолько прост, что его наверняка когда-нибудь стандартизируют.

Сначала следует определить тип, для которого будут переопределены глобальные функции `begin` и `end`.

```

1 template <typename T> struct reversion_wrapper {
2     T& iterable;
3 };

```

Теперь собственно переопределение это механическое упражнение, не требующее интеллектуальных усилий.

```

1 template <typename T>
2 auto begin (reversion_wrapper<T> w) {
3     return std::rbegin (w.iterable);
4 }
5
6 template <typename T>
7 auto end (reversion_wrapper<T> w) {
8     return std::rend (w.iterable);
9 }

```

И ключевой момент это определение адаптера. Всё что нужно это завернуть входной параметр в нужный тип, для которого уже всё переопределено.

```

1 template <typename T>
2 reversion_wrapper<T> reverse_cont (T&& iterable) {
3     return { iterable };
4 }

```

Это работает потому, что range-based обход не использует ничего, кроме `std::begin` и `std::end`.

Ещё одна распространённая стратегия адаптации это `inserters` – специальные итераторы, преобразующие запись по итератору во вставку в контейнер.

### 5.7.6 Валидность и инвалидация итераторов

Прежде, чем говорить об инвалидации, следует поговорить о валидности. Валидным может быть тератор или диапазон.

**Валидный итератор** это такой, который конформно поддерживает все операции для своей категории операторов. Например если валидный форвард-итератор увеличить на единицу, он будет указывать на следующий элемент (или станет невалидным). Если сделать такое с невалидным итератором, результат не будет определён.

**Валидный диапазон** состоит из двух валидных итераторов, причём второй достичим из первого.

```
1 istream_iterator<string> beg(ifstream("in.txt")), fin;
2 cross_copy (beg, fin,
3               ostream_iterator<string>(ofstream("out.txt")));
```

Диапазон, задаваемый в этом примере парой итераторов (`beg`, `fin`) невалиден, потому что невалиден итератор `beg`. Он является висячим итератором (по аналогии с висячим указателем), так как его файловый поток прекратил своё существование раньше его первого использования.

Итератор может оказаться невалидным по целому ряду причин:

- Он не инициализирован (т. н. сингулярные итераторы)
- Он подвис, т.е. ссылается на объект с истекшим сроком жизни
- Он указывает за пределы диапазона
- Он инвалидирован операциями над контейнером
- Это использованный итератор ввода

Сингулярные итераторы уже рассматривались ранее и иногда в них нет ничего плохого. Например в примере выше, итератор `fin` сингулярен, но вполне валиден. Но вот пример ситуации похуже.

```
1 list<string>::iterator lstit;
2 copy (vec.begin(), vec.end(), lstit);
```

Здесь сингулярный итератор не валиден и результат копирования не определён (UB). Аналог такого итератора – неинициализированный указатель.

Итераторы за границами диапазона делятся на два основных типа. Во-первых это итераторы, показывающие на `end()` или `rend()`, которые можно использовать но нельзя разыменовать (past-the-end итераторы)

```
1 list<int> lst;
2 auto past_end = lst.begin();
3 cout << *past_end << endl; // fail
4 lst.insert (past_end, 1); // ok
```

Во-вторых это итераторы, показывающие далеко после конца или раньше начала, с которыми нельзя делать ничего (out-of-range).

Итератор также может быть инвалидирован операциями над контейнером

```
1 vector<int> v = {11, 7, 5, 3, 2};  
2 auto vit = advance (v.begin(), 3);  
3 v.clear();  
4 cout << *vit << endl; // fail
```

Здесь итератор может инвалидирован и куда он указывает не определено. К сожалению, если итератор реализован как указатель, он будет куда-то указывать и ошибка может быть “тихой”.

Но самое плохое, даже не то, что она может быть тихой, а то, что она может быть плавающей. Простой пример.

```
1 vector<int> v = {2, 3, 5, 7, 11};  
2 auto vit = advance (v.begin(), 3);  
3 v.push_back(13);  
4 cout << *vit << endl; // ok???
```

Здесь итератор может быть инвалидирован, если capacity закончилось и случился `realloc`. А может и не быть.

Ниже подытожены правила базовой инвалидации для основных контейнеров на вставке и на удалении

## 1. Вставка

- `vector` – сохраняются все итераторы до точки вставки кроме случаев перевыделения, когда все инвалидированы
- `deque` – все итераторы всегда инвалидированы
- `list` – все итераторы сохраняются

## 2. Удаление

- `vector` – инвалидируются все итераторы после точки удаления, сохраняются до.
- `deque` – инвалидируются все итераторы при удалении из середины. При удалении из начала или конца, все итераторы сохраняются (кроме итератора на удаляемый).

- `list` – сохраняются все итераторы кроме итератора на удаляемый элемент

Ну и, пожалуй, самый неочевидный случай инвалидации это использованные `input`-итераторы

```
1 ifstream file("in.txt");
2 istream_iterator<string> beg(infile), end;
3 cross_copy (beg, end, ostream_iterator<string>(cout)); // ok
4 vector<string> vec (beg, end); // fail
```

Увы, проход второй раз по файлу невозможен. Хотя казалось бы и файл никуда не делся и вообще всё хорошо. Но нет.

## 5.8 Алгоритмы

*Computer science is no more about computers  
than astronomy is about telescopes*

– Edsger W. Dijkstra

Алгоритмы стандартной библиотеки это тот клей, который связывает контейнеры с итераторами. Контейнеры и итераторы полезны сами по себе, но вместе с алгоритмами они обретают законченность идеи и силу подлинной обобщённости.

Слово “алгоритм” сильно перегружено значениями. В контексте стандартной библиотеки, алгоритм это любая функция, которая оперирует на диапазоне, заданном итераторами.

### 5.8.1 Абстракция циклов

Начать рассмотрение алгоритмов следует с задачи, которая выглядит до обидного просто. Дан контейнер `cont`. Необходимо к каждому его элементу применить функцию `func`.

```
1 template <typename C, typename F>
2 void apply (C cont, F func)
3 {
4     // here goes some code
5 }
```

Программист с опытом на языках вроде С практически на автомате написал бы здесь нечто очевидное.

```
1 for (auto it = cont.begin(); it != cont.end(); ++it)
2     func (*it);
```

Человек, который дошёл до этой строчки, читая эту главу с начала, наверняка вспомнил бы также про range-based for (см. 5.7.1) и улучшил ситуацию, сократив код.

```
1 for (auto elt : cont)
2     func (elt);
```

Однако человек, действительно осознавший стандартную библиотеку, выбрал бы следующий (несколько странно выглядящий) вариант.

```
1 for_each (cont.begin(), cont.end(), func);
```

Здесь `std::for_each` это алгоритм (т.е. см. выше, функция) с сигнатурой

```
1 template<typename InputIterator, typename Func>
2 for_each (InputIterator fst, InputIterator lst, Func f);
```

С первого взгляда не ясно почему вызов функции вместо простого цикла это хорошая идея (и со второго). Тем не менее, это хорошая идея. Можно привести аргумент из C++17. Там вариант с `std::for_each` может быть переписан с учётом возможного параллелизма.

```
1 for_each (std::execution::par, cont.begin(), cont.end(), func);
```

Распараллелить так же просто цикл вряд ли получится. Кроме того, сама функция может быть реализована в стандартной библиотеке куда эффективней, потому что разработчик знает, что она делает. Можно рассматривать её как своего рода “специализацию” цикла. Видя в коде цикл общего вида, компилятор может применить лишь ограниченное количество оптимизаций, так как он не понимает и не может понимать что имелось в виду. Но каждый специализированный класс циклов задаваемый стандартной библиотекой служит лишь какой-то одной цели и может выполнять эту цель как эффективней.

В результате применение алгоритмов не только абстрагирует конкретные действия и улучшает читаемость кода, но может улучшить и его быстродействие. Такие абстракции (а все мы привыкли к тому, что абстракции не бесплатны) называются **абстракциями с отрицательной стоимостью**.

### 5.8.2 Копирование и суффиксы для алгоритмов

В стандартной библиотеке алгоритмов действительно много. Ещё в 98-м году их было около ста и с тех пор их число увеличивалось, сейчас приближаясь к 130 (точный подсчёт связан с некоторыми сложностями в методологии, например считать ли алгоритмом стандартной библиотеки доставшийся от С `qsort`). К счастью, нет никакой необходимости учить их все. В именовании алгоритмов есть своя система, понимая которую можно выучить не более чем 30-40 корней, получив все остальные даром.

Для примера, можно взять алгоритм `copy`.

```

1 template<class InputIterator, class OutputIterator>
2     OutputIterator
3 copy (InputIterator first, InputIterator last, OutputIterator
      result)

```

Его паттерн следующий:

```

1 // copy pattern
2 for (auto it = first; it != last; ++it)
3     *result++ = *it;

```

Но в его имени (как и в имени других алгоритмов) могут быть **суффиксы**. Кроме того, суффиксом может быть само слово `copy`. Основные суффиксы это:

- **if** (например `copy_if`) – означает, что алгоритм принимает дополнительный предикат и что действие будет выполнено, только если этот предикат истинный. В случае копирования, в его паттерн добавляется условие.

```

1 // copy_if pattern
2 for (auto it = first; it != last; ++it)
3     if (pred(*it))
4         *result++ = *it;

```

- **n** (например `copy_n`) – означает, что алгоритм принимает дополнительно количество раз, которые он должен выполнить своё действие.

```

1 // copy_n pattern
2 for (size_t i = 0; i != n; ++i)
3     *dst++ = *src++;

```

- **copy** (например `reverse_copy` означает, что обращённая последовательность должна быть размещена в новой памяти, а не разворнута на месте как в случае `reverse`). Ну и в целом для любого алгоритма означает, что он копирует свой результат в новое место, не используя старое.

Тут полезно разобрать несложный пример.

```

1 int myints[] = {2, 3, 5, 7, 11, 13, 17};
2 vector<int> myvector (7);

```

```

3
4 copy_n (myints, 7, myvector.begin());
5 copy (myvector.begin(), myvector.end(),
6       ostream_iterator<int>(cout, "\n"));
7
8 fill (myvector.begin(), myvector.end(), 0);
9 copy_if (myints, myints+7, myvector.begin(),
10          [] (int i){ return (i % 3) == 1; });
11 copy (myvector.begin(), myvector.end(),
12       ostream_iterator<int>(cout, "\n"));

```

**Вопрос к студентам:** что на экране?

Паттерны, которые нарисованы выше, конечно полезны. Но жизнь может отличаться и всегда полезно уметь увидеть в существующем коде паттерн.

Вот, например, код.

```

1 assert (cont.size() >= N);
2 auto it = cont.begin();
3 for (size_t idx = 0, idx != N; ++idx, ++it)
4     cout << *it << endl;

```

**Вопрос к студентам:** как переписать его алгоритмом?

Худшее, что тут может произойти: человек может начать видеть алгоритм `for_each` во всех циклах, в том числе и в приведённом выше. Это печальная ошибка. Разновидностей алгоритмов много и каждый специализирует цикл своим уникальным образом.

### 5.8.3 Общий обзор

Все алгоритмы можно (сравнительно произвольно) разбить на несколько основных категорий. Может показаться, что вообще-то хватило бы и первых двух: разбиение на модифицирующие и не модифицирующие алгоритмы выглядит исчерпывающим. Стандарт (C++17 28.1) делит алгоритмы на не изменяющие структуру последовательности (*non modifying*), изменяющие структуру последовательности (*mutating*) и похожие на сортировку (*sorting and related*) не считая категории-призрака алгоритмов, унаследованных от языка С. Это похоже на анекдот про три способа делать дело (правильный, неправильный и армейский), но такое усложнение имеет смысл.

Например мы договариваемся классифицировать всё по основному назначению алгоритма. Немодифицирующий алгоритм в этих терминах это просто информационный запрос к контейнеру, в то время, как модифицирующий нужен именно для того, чтобы изменить структуру контейнера. В этом смысле, несомненно, `sort` это модифицирующий алгоритм, а `is_sorted` – нет. Но есть нечто большее, связывающее эти два алгоритма: оба они относятся к сортировке и их, поэтому, можно выделить в специальную группу вместе с другими, относящимися к сортировке и разбиению алгоритмами, как модифицирующими, так и не модифицирующими.

Автор этих лекций в итоге остановился на шести основных категориях.

### Немодифицирующие

- `all`, `any`, `none`
- `for_each` (`n`)
- `find` (`if`), `count` (`if`)
- `search`, `find_end`, `find_first_of`
- `mismatch`
- `adjacent_find`
- `min` (`element`), `max` (`element`), `minmax` (`element`)
- `clamp`
- `equal`, `lexicographical_compare`

Собранные в этой секции алгоритмы делают запрос самого общего вида. Начиная от `all` и заканчивая `find_first_of` это всё поиск на несортированных контейнерах. При этом даже если контейнер сортированный, они будут работать с ним как с несортированным. Несколько особняком стоит `for_each` потому что он, строго говоря, ничего не запрашивает. Но принцип его работы такой же, как у поиска. Только он “находит” все элементы контейнера. При объяснении на лекциях, наибольший интерес вызывали `adjacent_find` и `clamp`

## Модифицирующие

- `copy (if)(n) (backward)`, `move (backward)`
- `swap`, `swap_ranges`, `iter_swap`
- `transform`
- `replace (copy)(if)`
- `fill (n)`
- `generate (n)`
- `remove (copy)(if)`
- `unique (copy)`
- `reverse, rotate (copy)`
- `shuffle`
- `sample`

Здесь тоже в целом всё понятно. Обычно некое смущение вызывает алгоритм

## Сортировка и бинарный поиск

- `is_partitioned`, `is_sorted`
- `partition`, `stable_partition`
- `sort`, `partial_sort`, `stable_sort`
- `nth_element`
- `binary_search`
- `lower_bound`
- `upper_bound`
- `equal_range`
- `merge`
- `includes`
- set operations: `set_difference`, `set_intersection`, etc...

## Слияния и кучи

- `is_heap`
- `make_heap`
- `push_heap`
- `pop_heap`

## Численные

- `is_permutation`
- `next_permutation`
- `prev_permutation`
- `accumulate, reduce`
- `iota`
- `partial_sum, exclusive_scan, inclusive_scan`
- `adjacent_difference`
- `inner_product`

В этот обзор не вошла работа с неинициализированной памятью (такие функции как `uninitialized_default_construct`) и наследство стандартной библиотеки С. Легко подсчитать, что здесь не больше 50 пунктов, которые и определяют основное ядро алгоритмов, вместе с суффиксами и всякой экзотикой.

### 5.8.4 Идиома `erase-remove`

Удаление элементов из контейнера при работе с итераторами имеет свои контринтуитивные особенности. Формально для удаления служит алгоритм `std::remove` и здесь может возникнуть искушение воспользоваться им что называется “в лоб”.

```
1 std::vector<int>& vec = myNumbers;
2 std::remove(vec.begin(), vec.end(), number_in);
```

Увы это не работает и не может работать. Ни у какого алгоритма не может быть достаточно информации чтобы по паре входных итераторов произвести честное удаление из обобщенного контейнера, потому что это потребовало бы от них по сути прямого управления памятью контейнера. Что происходит на самом деле проиллюстрировано ниже:

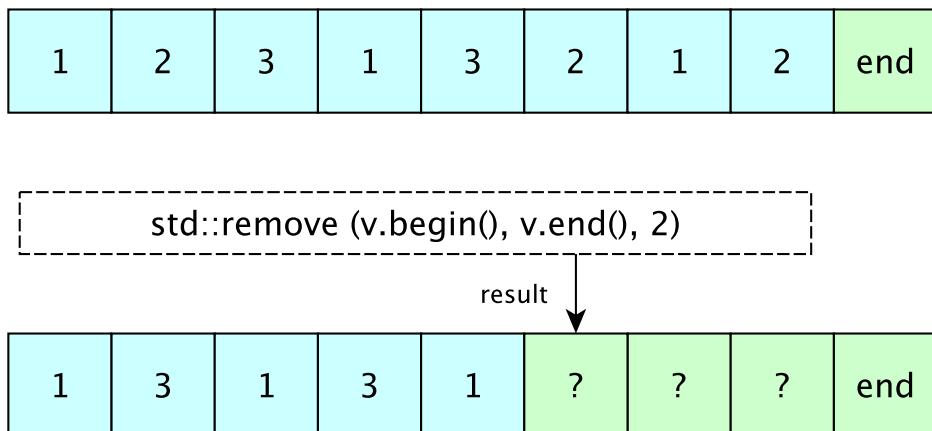


Рис. 5.23: Идиома erase-remove

Произошел сдвиг всех элементов, попадающих под условие удаления в хвост контейнера. Далее чтобы действительно очистить контейнер, нужно вызвать его метод `erase` например так:

```

1 vector<int>& vec = myNumbers;
2 vec.erase(remove(vec.begin(), vec.end(), number_in), vec.end())
;
```

Этот код записывается слитно очень часто, так что такие паттерны, как использование этой идиомы, опытный разработчик должен распознавать в коде “на лету”.

Мало того, опознавать идиому `erase-remove` нужно также в тех случаях, когда там вообще нет `remove`. Мало ли алгоритмов удаляют данные. Вот, например, `unique`

```

1 vector v = {1, 42, 2, 42, 3, 42, 4};
2 sort (v.begin(), v.end());
3 v.erase (unique (v.begin(), v.end()), v.end());
```

К счастью, в стандартной библиотеке таких алгоритмов мало (собственно `remove`, `remove_if` и `unique`), но в пользовательском коде может встретиться всякое.

### 5.8.5 Абстракция циклов

TODO: перенести со слайдов и поставить ссылку на Парента [36]

### 5.8.6 Трансформации

В функциональных языках люди очень любят функции вроде `map`, которые абстрагируют применение функтора к последовательности с рождением другой последовательности. В терминах какого-нибудь Haskell, это нечто вроде:

```
map :: (a -> b) -> [a] -> [b]
```

Для C++ роль такой функции выполняет `transform`. Он до некоторой степени похож на `for_each` до степени смешения. Поэтому важно понимать, что у этих функторов разные соответствующие им паттерны циклов.

Например обратить знак у последовательности целых чисел легко как с помощью одного, так и с помощью другого, но паттерн применения принципиально разный.

```
1  vector<int> v = {2, 3, 5, 7, 11, 13};
2
3  // Option 1.
4  for_each(v.begin(), v.end(), [] (auto& i) { i = -i; } );
5
6  // Option 2.
7  transform (v.begin(), v.end(), v.begin(), [] (auto i) {
8      return -i; } );
```

Из этих двух вариантов, предпочтителен `transform`, так как ничего не утекает по косвенности.

Но главное их отличие в другом: в отличии от `for_each`, `transform` не гарантирует порядка выполнения. Программист указывает трансформировать последовательность, но не говорит как конкретно это сделать.

TODO: в слайдах у меня тут функторы против лямбд. Должно ли это быть здесь или вынести это – вопрос.

На примере `transform` можно разобрать неожиданные проблемы, возникающие при попытке применять контейнеры с перегруженными функциями. Общеизвестна функция `toupper`, приплывшая в C++ из стандартной библиотеки языка С. Попытка применить её как аргумент для трансформации строчки натыкается на небольшую проблему.

```
1 string s="hello";
2 transform(s.begin(), s.end(), s.begin(), std::toupper);
```

Небольшая проблема (с сообщением об ошибке примерно на два экрана) сводится к тому, что этот вариант не компилируется. Не компилируется он, как и было сказано, поскольку, приплыв из стандартной библиотеки С, `toupper` раздвоилась (такое бывает). C++ знает её уже в двух вариантах:

```
1 charT toupper(charT) const;
2 const charT* toupper(charT* low, const charT* high) const;
```

Вызов `transform` не содержит достаточного контекста для разрешения перегрузки. Выходов (простых) тут два. Во-первых можно разрешить перегрузку вручную явным приведением типа. Во-вторых можно создать контекст, обернув всё в лямбда функцию.

```
1 transform(s.begin(), s.end(), s.begin(),
2           static_cast<int(*)(int)>(std::toupper));
3
4 transform(s.begin(), s.end(), s.begin(),
5           [] (auto x){ return std::toupper(x); });
```

Эстетика радуется второму варианту, но формально первый вариант создаёт на одну косвенность меньше, так что кто знает.

TODO: `transform` и `move` семантика. Требует рассмотрения `move` итераторов, а это адаптер.

### 5.8.7 Бинарный поиск и его варианты

До сих пор рассматривались алгоритмы, которые не накладывали никаких неявных ограничений на диапазон (кроме валидности, но валидность нужна всегда). Но не все алгоритмы столь няшно доброжелательны. Некоторые алгоритмы, самым известным из которых является бинарный поиск, требуют отсортированного интервала и работают на неотсортированном непредсказуемо.

```

1 vector<int> v = {81, 9, 54, 36, 27, 63, 18, 72, 45};
2 sort (v.begin(), v.end());
3 if (binary_search (v.begin(), v.end(), 37) {
4     assert (0 && "Oh, no!");
5 }
```

Как видно из примера выше, алгоритм, который имеет многообещающее название `binary_search`, на самом деле ничего не находит. Он возвращает логическое значение – есть элемент в контейнере или его там нет.

### 5.8.8 Алгоритмический подход к перестановкам

Методы абстракции, рассмотренные в этой главе, простираются далеко за пределы стандартной библиотеки. Вовсе не обязательно иметь стандартные контейнеры, итераторы и так далее, чтобы использовать эти подходы.

Для примера, здесь будет рассмотрена такая далёкая от стандартной библиотеки вещь, как перестановки.

Мы можем кодировать перестановки любых объектов как циклические перестановки. Простейший цикл это (1 2) означает, что 1 переходит в 2 и 2 переходит в 1, то есть 1 и 2 меняются местами. Цикл (2 3 1) означает  $2 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 2$ .

Очевидно, что (1 2 3) == (2 3 1) == (3 1 2). Изо всех равных циклов можно выбрать один у которого наименьший элемент стоит первым в записи цикла. Это будет (1 2 3) и этот цикл называется нормальной формой записи своего класса эквивалентности.

**Вопрос к студентам:** какая будет нормальная форма для (4 2 1 3)?

Думать о перестановках в терминах циклов несколько необычно. Обычно о перестановках думают как о двух столбиках, первый из которых задаёт начальную строчку, а второй строчку после перестановки.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 9 & 2 & 3 & 1 & 7 & 6 & 8 & 5 & 4 \end{pmatrix}$$

Вся информация при этом сконцентрирована в нижнем столбике, поскольку верхний тривиален.

Необходимо написать функцию, которая переводит нижний столбик

записи перестановки в циклическую форму. В данном случае ответом будет  $(1\ 9\ 4)(2\ 3)(5\ 7\ 8)(6)$ .

Для начала нужно ответить на вопрос какая у этой функции будет сигнатура.

Пожалуйста прежде чем читать дальше выпишите самостоятельно сигнатуру которую вы предлагаете.

Если вы выписали нечто вроде

```

1 // creates array of loops from permutation given by table
2 // say: a, g, [d, c, e, g, b, f, a]
3 // gives: [(a, d, g), (b, c, e), (f)]
4 template <typename T>
5 void create_loops(T start, T fin, const vector<T>& table,
6                   vector<PermLoop<T>>& out);

```

То поздравляю, вы ничего не поняли про алгоритмы.

Во-первых кодировать диапазон как  $(start, fin)$  явными параметрами крайне избыточно. Лучше предположить, что тип  $T$  это домен у которого уже есть и  $T::start$  и  $T::fin$  (или, скажем  $perm::start(T)$  и  $perm::fin(T)$ ).

Во-вторых ограничение входного типа вектором и, хуже того, всем вектором сразу, делает эту функцию далёкой от обобщённости. Но казалось бы мы же уже изучали обобщённые алгоритмы. Почему не сделать эту функцию берущей итераторы.

Близкий к совершенству ответ выглядит так:

```

1 template <typename RandIt, typename OutIt>
2 void create_loops(RandIt tbeg, RandIt tend, OutIt lbegin);

```

Теперь эта функция – обобщённый алгоритм. Можно реализовать её, но это техническое упражнение, а здесь хочется сконцентрироваться на идеях.

Теперь применение циклов перестановок. Применение это метод в классе  $PermLoop<T>$  и перестановка может применяться как к числу, так и к вектору. Применение  $(1\ 2)$  к числу 1 даёт 2. Применение  $(1\ 3)$  к вектору  $[1\ 2\ 3\ 4\ 5\ 6]$  даёт в итоге  $[3\ 2\ 1\ 4\ 5\ 6]$ .

Метод  $apply$  для числа это простое применение цикла, использующее в частности другие алгоритмы STL, например  $find$ .

```

1 template <typename T> T PermLoop<T>::apply (T x) const {

```

```

2   auto it = find(loop_.begin(), loop_.end(), x);
3   if (it == loop_.end()) return x;
4   auto nxt = next(it);
5   if (nxt == loop_.end()) return *loop_.begin();
6   return *nxt;
7 }
```

Сигнатура метода `apply` для таблицы это снова обобщённый алгоритм, берущий пару итераторов и применяющий перестановку к диапазону.

```

1 template <typename T>
2 template <typename RandIt>
3 void PermLoop<T>::apply(RandIt tbeg, RandIt tend) const;
```

Имя типа и тут и в предыдущем примере подсказывает, что требуются итераторы с произвольным доступом.

Теперь настало время заняться действительно интересной вещью: перемножить перестановки. До сих пор перестановки комбинировались когда они были независимы, например  $(1\ 2\ 4)(3\ 5)$ .

Но что мешает скомбинировать зависимые перестановки?

Запись  $(1\ 2)(2\ 3)$  означает  $1 \rightarrow 2$ ,  $2 \rightarrow 1$  и далее  $2 \rightarrow 3$ ,  $3 \rightarrow 2$ . В итоге получается, что 1 переходит в 2 а потом в 3, значит 1 переходит в 3, 3 в 2, а 2 в 1.

В циклической записи  $(1\ 2)(2\ 3) = (1\ 3\ 2)$

Более сложный пример:  $(1\ 3\ 2)(1\ 2\ 4)(1\ 4\ 3\ 2) = (1\ 2)(3)(4)$

Какую сигнатуру должна иметь функция перемножения (упрощения массива циклов) перестановок?

И опять есть неправильный, но соблазнительный вариант

```

1 template <typename T>
2 void simplify_loops (vector<PermLoop<T>> &input);
```

**Вопрос к студентам:** самостоятельно перечислить все его проблемы.

Правильный вариант в алгоритмическом стиле снова использует итераторы.

```

1 template <typename RandIt, typename OutIt>
2 void simplify_loops (RandIt tbeg, RandIt tend, OutIt lbeg);
```

Реализация этого алгоритма происходит в терминах двух уже определённых ранее алгоритмов, комбинируя их благодаря механизму итераторов

```
1 template <typename RandIt, typename OutIt>
2 void simplify_loops (RandIt tbeg, RandIt tend, OutIt lbeg) {
3     using T = typename std::decay<decltype(*tbeg)>::type::
4         value_type;
5     vector<T> table(T::fin - T::start + 1, T::start);
6     iota(table.begin(), table.end(), T::start);
7     for (auto loopit = make_reverse_iterator(tend);
8          loopit != make_reverse_iterator(tbeg);
9          ++loopit)
10    loopit->apply(table.begin(), table.end());
11    create_loops(table.begin(), table.end(), lbeg);
12 }
```

Это очень важный код, над ним можно немного помедитировать. Он показывает как в максимально далёкой от стандартной библиотеки предметной области действуют те же подходы, что применялись Степановым и при разработке библиотеки. Именно это – подлинная мощь библиотеки и мышления в её терминах.

## 5.9 Ассоциативные контейнеры

Ранее в приложении к алгоритмам изучались перестановки (см. 5.8.8). Перестановки образуют группы.

Группа это множество с транзитивной операцией над множеством, такой, что в множестве относительно этой операции есть единичный элемент и у каждого элемента есть обратный. Перемножение перестановок является групповой операцией над множеством перестановок. Единичная перестановка это перестановка, оставляющая всё на местах.

Группа может быть сгенерирована, когда задан только единичный элемент и несколько генераторов, а остальные элементы группы получаются последовательными применениями генераторов к новым элементам.

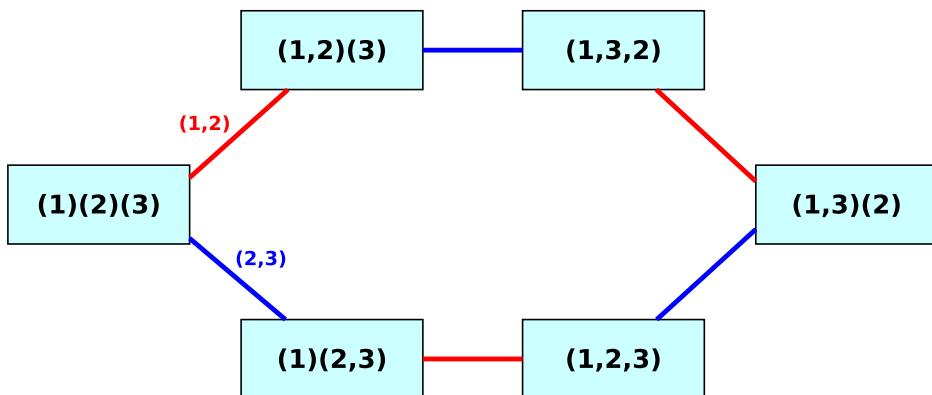


Рис. 5.24: Группа перестановок  $S(3)$

На (рис. 5.24) приведена группа перестановок  $\text{Sym}(3)$  с генераторами  $(1, 2)$  и  $(2, 3)$ . Первый показан красными линиями, второй – синими. По линиям можно двигаться в обоих направлениях, так как умножение на цикл длиной два второй раз это просто возврат к предыдущему значению.

Орбитой элемента в группе называются все значения, в которые его переводят элементы группы. Скажем орбита элемента 1 в  $\text{Sym}(3)$  это  $(1, 2, 3)$  как это показано на (рис. 5.25).

Вычисление орбиты естественно производить методом транзитивного замыкания. Идёт обход неявного дерева с отсечением ветвей, как показано на (рис. 5.26)

Если написать процедуру, делающую такой обход в обобщённом сти-

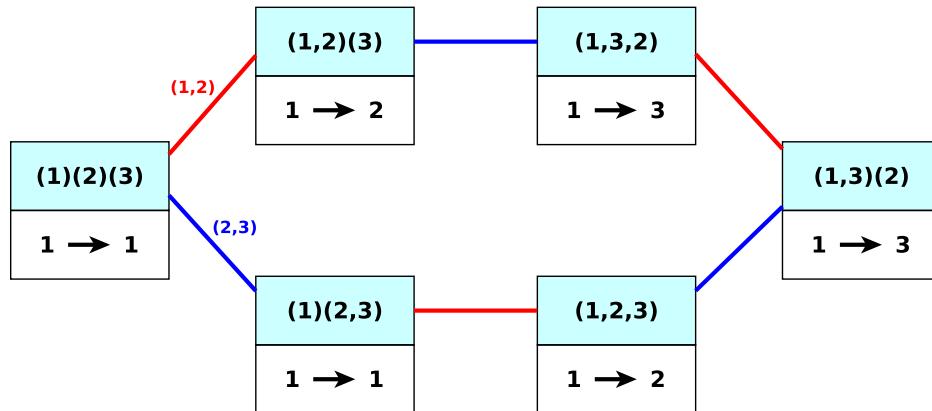
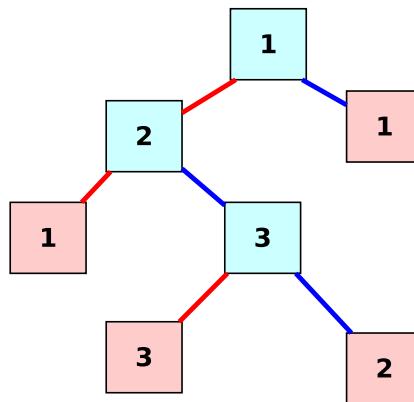
Рис. 5.25: Орбита элемента 1 в  $S(3)$ 

Рис. 5.26: Обход группы и вычисление орбиты

ле, то она будет выглядеть как-то так.

```

1 template<typename T, typename RandIt>
2 auto simple_orbit(T num, RandIt gensbeg, RandIt gendsend) {
3     vector<T> orbit, next = {num};
4     while (!next.empty()) {
5         vector<T> tmp {};
6         orbit.insert(orbit.end(), next.begin(), next.end());
7         for (const auto &elem : next)
8             for (auto igen = gensbeg; igen != gendsend; ++igen)
9                 if (auto newelem = igen->apply(elem);
10                     find(orbit.begin(), orbit.end(), newelem) == orbit.
11                         end())
12                     tmp.push_back(newelem);

```

```

12     next.swap(tmp);
13 }
14 orbit.erase(unique(orbit.begin(), orbit.end()), orbit.end());
15 return orbit;
16 }
```

**Вопрос к студентам:** оцените алгоритмическую сложность этой процедуры?

Даже если вы не можете дать ответ на этот вопрос (в конце концов это лекции по C++ а не по алгоритмам), хорошему программисту всё равно очевиден источник основных проблем с производительностью в этой процедуре. Если вам он не очевиден, не расстраивайтесь, вы просто несколько нейроотличны, возьмите профилировщик.

Разумеется, линейный поиск внутри трёх вложенных циклов это вот прямо то чего мы всегда хотели от жизни. Можно ли улучшить? Наверное да. Если вектора сортированные, то поиск мог бы быть бинарным.

```

1 template<typename T, typename RandIt>
2 auto simple_orbit(T num, RandIt gensbeg, RandIt gendsend) {
3     vector<T> orbit, next = {num};
4     while (!next.empty()) {
5         vector<T> tmp {};
6         orbit.insert(orbit.end(), next.begin(), next.end());
7         sort(orbit.begin(), orbit.end());
8         for (const auto &elem : next)
9             for (auto igen = gensbeg; igen != gendsend; ++igen)
10                 if (auto newelem = igen->apply(elem));
11                     !binary_search(orbit.begin(), orbit.end(), newelem))
12                     tmp.push_back(newelem);
13         next.swap(tmp);
14     }
15     orbit.erase(unique(orbit.begin(), orbit.end()), orbit.end());
16     return orbit;
17 }
```

Теперь сложность понизилась до  $O(N * \ln(N) * r)$  но печальная необходимость сортировать вектор каждый раз даёт довольно большую константу при этой асимптотике.

Было бы здорово, если бы был контейнер, который сам поддерживал свою отсортированность и позволял быструю вставку элементов, при этом поддерживая их уникальность. И такой контейнер есть. Это мно-

жество `std::set`.

### 5.9.1 Множества

В стандартной библиотеке классы множества и мультимножества определены следующим образом

```

1 template <class Key, class Compare = less<Key>,
2         class Allocator = allocator<Key>>
3 class set;
4
5 template <class Key, class Compare = less<Key>,
6         class Allocator = allocator<Key>>
7 class multiset;
```

На базовом уровне можно думать о множестве как о всегда отсортированном массиве уникальных элементов, не вдаваясь в то, чем оно на самом деле является.

Множество хранит уникальные элементы

```

1 set<int> s = {67, 42, 141, 23, 42, 106, 15, 50};
2 for (auto elt : s)
3     cout << elt << endl;
```

Здесь дважды упомянуто число 42. Ничего не сломается, но на экране будет

15, 23, 42, 50, 67, 106, 141

С помощью множеств, функция `simple_orbit` может быть переписана гораздо проще.

```

1 template<typename T, typename RandIt>
2 auto simple_orbit(T num, RandIt gensbeg, RandIt gendsend) {
3     set<T> orbit, next = {num};
4     while (!next.empty()) {
5         set<T> tmp {};
6         orbit.insert(next.begin(), next.end());
7         for (const auto &elem : next)
8             for (auto igen = gensbeg; igen != gendsend; ++igen)
9                 if (auto newelem = igen->apply(elem); orbit.count(
newelem) == 0)
```

```

10         tmp.insert(newelem);
11     next.swap(tmp);
12 }
13 return orbit;
14 }
```

Простой замер производительности показывает, что речь идёт о выигрыше в разы по сравнению с сортированными векторами.

Множества создают упорядочение своих элементов.

```

1 set<int> s = {67, 42, 141, 23, 42, 106, 15, 50};
2 auto itb = s.lower_bound(30);
3 auto ite = s.upper_bound(100);
```

Теперь можно итерировать в интервале [30, 100) независимо от того есть ли в точности во множестве такие элементы.

```

1 for (auto it = itb; it != ite; ++it)
2   cout << *it << endl;
```

**Вопрос к студентам:** что будет выведено на экран?

Второй шаблонный параметр подсказывает, что множества позволяют задавать порядок итерирования

```

1 set<int, greater<int>> s = {67, 42, 141, 23, 42, 106, 50};
2 auto itb = s.lower_bound(30);
3 auto ite = s.upper_bound(100);
```

Увы, теперь интервал некорректен. С точки зрения предиката `greater` число 100 предшествует числу 30 и таким образом итератор `ite` не достижим из `itb` (см. 5.7.6).

Запись, вроде приведённой ниже

```

1 auto itb = s.lower_bound(100);
2 auto ite = s.upper_bound(30);
```

Создаёт валидный диапазон для множества с предикатом `greater`. Увы, это никак не контролируется на этапе компиляции.

Интересный вопрос: что будет если упорядочить по критерию “меньше или равно” например таким образом

```
1 set<int, less_equal<int>> s = {67, 42, 141, 23, 42, 106, 50};
```

В получившемся множестве тогда окажется два элемента со значением 42. Это нарушает инвариант контейнера и может привести к странным сложно отлавливаемым багам.

Это наводит на мысль, что не каждый предикат подходит для упорядоченного контейнера. И действительно, общая концепция, которой можно проверить годность предиката называется strict weak ordering. Она включает

1. Антисимметричность  $\text{pred}(x, y) \Rightarrow \neg\text{pred}(y, x)$
2. Транзитивность  $\text{pred}(x, y) \wedge \text{pred}(y, z) \Rightarrow \text{pred}(x, z)$
3. Иррефлексивность  $\neg\text{pred}(x, x)$
4. Транзитивность эквивалентности  $\text{eq}(x, y) \equiv \neg\text{pred}(x, y) \wedge \neg\text{pred}(y, x)$   
 $\text{eq}(x, y) \wedge \text{eq}(y, z) \Rightarrow \text{eq}(x, z)$

Она же распространяется на предикаты в алгоритмах сортировки и т.д.

**Домашняя наработка:** пусть  $(a + ib < c + id) \Leftrightarrow (a < c) \wedge (b > d)$  является ли это strict weak ordering для комплексных чисел?

Концепция strict weak ordering важна даже в мультимножествах, где казалось бы нет проблемы хранить два одинаковых элемента. Майерс [6] приводит красивый контрпример

```

1 multiset<int, less_equal<int>> s;
2 s.insert(10); // insert 10A
3 s.insert(10); // insert 10B

```

Теперь `equal_range` для 10 вернёт пустой интервал, поскольку искомое значение 10 не будет опознано как равное двум уже вставленным.

### 5.9.2 Модификация множеств

Простейшая модификация элемента множества на месте не работает и работать не должна

```

1 *s.begin() = 3; // error: assignment of read-only location

```

В самом лучшем случае это ошибка компиляции. В худшем случае ваша реализация не поддерживает хранения константных ключей в множестве и у вас проблемы.

Дело в том, что внутренне множество скорее всего будет представлено какой-либо разновидностью сбалансированного дерева. Изменение ключа нарушает балансировку и не должно производиться иначе, чем методами класса.

По этой же причине для множеств не работает идиома `erase-remove`

```
1 set<int> s = {1, 2};
2 s.erase(remove(s.begin(), s.end(), 1), s.end());
```

В лучшем случае это тоже ошибка компиляции. Удалять из множеств, сохраняя балансировку можно только через их внутренний метод `erase`.

```
1 s.erase(1);
```

Правда тут тоже есть проблема. Используя такое удаление в цикле слишком легко выстрелить себе в ногу.

```
1 for (auto it = s.begin(); it != s.end(); ++it)
2   if ((*it < 100) && (*it > 30))
3     s.erase(it);
```

Это очень плохой код, никогда так не делайте. Дело в том, что удаляя элемент вы инвалидируете итератор на этот элемент. Ну и всё. Далее итерация по невалидному итератору заинтересует вас туда, куда Макар телят итерировал.

Решение этой проблемы в C++98 выглядело как-то вот так

```
1 for (auto it = s.begin(); it != s.end();)
2   if ((*it < 100) && (*it > 30))
3     s.erase(it++);
4 else
5   ++it;
```

Начиная с C++11, метод `erase` в множествах возвращает итератор на валидный элемент и этим можно наслаждаться.

```
1 for (auto it = s.begin(); it != s.end();)
2   if ((*it < 100) && (*it > 30))
3     it = s.erase(it);
4 else
5   ++it;
```

Не знаю, впрочем, стало ли от этого всем сильно легче.

Если вам (с чего был начат этот параграф) всё-таки нужно заменить элемент в множестве, то методами множества законно делать это только стирая старый элемент и добавляя новый.

```
1 auto it = s.find(1);
2 if (it != s.end()) {
3     s.erase(it);
4     s.insert(3);
5 }
```

Именно эта чувствительность множеств – тот факт, что они хранят только ключи и всё, делает их в некоторых контекстах очень плохим выбором для контейнера.

### 5.9.3 Отображения

Если внимательно рассмотреть орбиту элемента 1 в  $\text{Sym}(3)$  (рис. 5.25), можно заметить, что из шести перестановок группы ровно две оставляют 1 на месте, две превращают его в 2 и две превращают его в 3. Эта симметрия не случайна и за ней лежит некая математическая теория, выходящая за пределы этого конспекта. Но кажется очевидным, что для любого элемента орбиты должен быть элемент группы (называемый также coset representative) который сдвигает элемент, для которого ищется орбита в элемент орбиты.

Пусть стоит задача найти все такие элементы.

Имея только множества, лучшее, что можно придумать это множество пар (элемент, перестановка).

```

13     [newelem](auto&& elt) { return (newelem == elt.
14         first); }) == orbit.end())
15     tmp.emplace(newelem, product(curgen, *igen));
16     next.swap(tmp);
17 }
18 }
```

Но у этого подхода снова есть проблема, при чём та же самая: линейный поиск. Причём тут сложно что то придумать: в контейнере `set` есть метод `find`, но нет и не может быть `find_if`.

**Вопрос к студентам:** вы понимаете что не так с `find_if` для `set`.

Для решения таких вопросов: хранения ключа и чего-то вроде значения по этому ключу в стандартной библиотеке существуют отображения и мультиотображения. Они объявлены следующим образом

```

1 template <class Key, class T, class Compare = less<Key>,
2 class Allocator = allocator<pair<const Key, T>>>
3 class map;
4
5 template <class Key, class T, class Compare = less<Key>,
6 class Allocator = allocator<pair<const Key, T>>>
7 class multimap;
```

Здесь можно особо отметить класс для которого аллокатор. В отличии от `set`, здесь нам явно говорят что именно менять нельзя.

С использованием отображений, решение становится простым и ясным

```

1 template<typename T, typename RandIt>
2 auto orbit_maps(T num, RandIt gensbeg, RandIt gendsend) {
3     map<T, Permutation<T>> orbit, next {{ num, {} }};
4     while (!next.empty()) {
5         map<T, Permutation<T>> tmp {};
6         orbit.insert(next.begin(), next.end());
7         for (auto&& [elem, curgen] : next)
8             for (auto igen = gensbeg; igen != gendsend; ++igen)
9                 if (auto newelem = igen->apply(elem); orbit.find(
10                     newelem) == orbit.end())
11                     tmp.insert({newelem, product(curgen, *igen)}));
12         next.swap(tmp);
13 }
```

```

12     }
13     return orbit;
14 }
```

У отображения, метод `find` ищет по ключам с логарифмической сложностью. Если мы оцениваем перемножение перестановок дешевле, чем проверку, что перестановка существует, можно убрать `if` так как если в отображении ключ уже существует, вставлено ничего не будет. Но мои замеры производительности показывают, что для огромных перестановок это плохая идея, поскольку ключ это что-то вроде целого числа, а значение тут что-то вроде вектора векторов.

#### 5.9.4 Модификация отображений

В приведённом выше листинге есть места, вызывающие подозрение. В первую очередь это вставка.

```
1 tmp.insert({newelem, product(curgens, *igen)});
```

Здесь она сделана через `insert`, но ведь есть же `emplace`, его не может не быть.

```
1 tmp.emplace(newelem, product(curgens, *igen));
```

Это действительно работает, и, казалось бы, этот метод экономит однокрайне тяжёлое копирование одной крайне тяжёлой пары. Увы, это не так. Майерс в [7] посвятил этому целый совет 42, поэтому не стоит подробно останавливаться здесь. Вкратце: для ассоциативных контейнеров из-за того, что у них при вставке есть также поиск места для вставки в, условно говоря, дерево, узел всё равно должен быть создан довольно рано. Метод `emplace` просто заносит его создание внутрь и имеет по сравнению с `insert` не лучшую производительность и проблемы с исключениями. Об этом методе (в чистом виде) для ассоциативных контейнеров можно просто забыть.

Третий вариант специфичен только для отображений и предполагает использование квадратных скобок.

```
1 tmp[newelem] = product(curgens, *igen);
```

Использование квадратных скобок для выражения семантики “обновление-или-вставка” – очень странное проектное решение. Его нет ни в множествах ни даже в мультиотображениях. В самих отображениях такое ис-

пользование квадратных скобок плохо видно в коде и иногда приводит к странным моментам.

```
1 cout << m[x] << endl;
```

Здесь если `m` это `map`, то вообще-то ключ `x` может быть вставлен в отображение!

Впрочем если обновление действительно нужно, использование квадратных скобок эффективно и обоснованно, так что его следует обдумать. В рассматриваемом случае речь не идёт об обновлении, поэтому именно для групп мы его отмечаем, но вообще квадратные скобки несмотря на их минусы могут быть удобны.

И, наконец, лучше всего себя показывает четвертый вариант: `emplace` с указанием места вставки, так называемый `emplace_hint`.

Чтобы его мотивировать, можно рассмотреть не одну строчку а две.

```
1 if (auto newelem = igen->apply(elem); orbit.find(newelem) ==
     orbit.end())
2     tmp.insert({newelem, product(curgen, *igen)});
```

И заметить, что здесь дважды делается одна работа: элемент ищется методом `find`, а потом ещё раз ищется при вставке через `insert` во время поиска позиции для вставки.

```
1 auto newelem = igen->apply(elem);
2 auto it = orbit.lower_bound(newelem);
3 if (it == orbit.end() || it->first != newelem)
4     tmp.emplace_hint(it, newelem, product(curgen, *igen));
```

Использование `emplace_hint` как показано выше позволяет не делать лишней работы, а искать один раз и искать целенаправленно позицию для возможной вставки с использованием `lower_bound`.

Стандарт C++17 принёс в модификацию отображений два новшества. Во-первых вытащить из дерева целую ноду, при этом удалив её полностью и ребалансировав дерево можно с помощью `extract`.

```
1 map<int, string> m1 = {{1, "sator"}, {2, "tenet"}, {3, "nope"}};
2 auto extval = m1.extract(3);
```

О том, какого типа получается `extval` лучше даже не задумываться: в стандарте он помечен как полностью Implementation defined, вплоть до размера и состава полей. Всё, что от него требуется это определённый набор методов, среди которых самые полезные это `key()` и `value()`

```
1 cout << extval.key() << endl; // prints 3
```

Во-вторых соединить два отображения в одно теперь можно с помощью `merge`

```
1 map<int, string> m1 = {{1, "sator"}, {2, "tenet"}};
2 map<int, string> m2 = {{2, "nothing"}, {3, "arepo"},
3                           {4, "opera"}, {5, "rotas"}};
4 m1.merge(m2);
```

При этом все элементы `m2` по одному вынимаются (например через `extract`) и вставляются в `m1`. Метод несколько эффективней, чем делать то же самое простым циклом или даже алгоритмом.

### 5.9.5 Словари

Простая орбита с использованием множества всё-таки не слишком хороша.

```
1 for (const auto &elem : next)
2     for (auto igen = gensbeg; igen != gensend; ++igen)
3         if (auto newelem = igen->apply(elem);
4             orbit.count(newelem) == 0)
5                 tmp.insert(newelem);
```

В главном цикле притаился метод `orbit.count`, имеющий, между прочим, сложность  $O(\ln(N))$ . Этот логарифмический множитель выглядит болезненно и от него хотелось бы избавиться. Обычно для этого надо чем-то пожертвовать.

Основная идея, которую приносят **словари** это пожертвовать в данном случае упорядоченностью.

Словари определены в стандарте как четыре класса: неупорядоченные множества, мульти множество, отображение и мультиотображение.

Неупорядоченное множество.

```
1 template<
2     class Key, class Hash = hash<Key>,
3     class KeyEqual = equal_to<Key>,
4     class Allocator = allocator<Key>
5 > class unordered_set;
```

Почти аналогично: неупорядоченное отображение.

```

1 template<
2     class Key, class T, class Hash = hash<Key>,
3     class KeyEqual = equal_to<Key>,
4     class Allocator = allocator<pair<const Key, T>>
5 > class unordered_map;

```

Мульти множество и мультиотображение по сути аналогичны.

С использованием неупорядоченного множества, вычисление простой орбиты теперь можно переписать более вдохновляющим образом.

```

1 template<typename T, typename RandIt>
2 auto simple_orbit(T num, RandIt gensbeg, RandIt gendsend) {
3     unordered_set<typename T::type> orbit;
4     vector<T> next = {num};
5     while (!next.empty()) {
6         unordered_set<typename T::type> tmp {};
7         orbit.insert(next.begin(), next.end());
8         for (const auto &elem : next)
9             for (auto igen = gensbeg; igen != gendsend; ++igen)
10                 if (auto newelem = igen->apply(elem));
11                     orbit.count(newelem) == 0) // O(1) +
12                         tmp.insert(newelem);
13         next.swap(tmp);
14     }
15     return orbit;
16 }

```

Теперь сложность, вероятно, улучшилась, поскольку дерево было заменено на хэш. Но у `unordered_set` нет методов `lower_bound` и `upper_bound`

Пляски с `typename T::type` связаны с тем, что `T` это что-то вроде `IDom<unsigned, 1, 7>` и у него просто лень протягивать хэш и равенство, а проще воспользоваться нижележащим примитивным типом. Но вообще, конечно, конечно и добавить, чуть позже об этом пойдёт особый разговор (см. 5.9.6).

Общий гайдлайн такой: если всё, что нужно это ассоциативный массив и его упорядоченность никак не используется, то механическая замена `set` на `unordered_set` это первое, что следует попробовать.

Самый частый источник проблем при подобного рода замене это требования к интерфейсу типа. Вместо `std::less` теперь по умолчанию

идут `std::hash` и `std::equal_to`. В общем случае сделать оператор “меньше” проще, чем сравнение и хэш.

### 5.9.6 Переход к неупорядоченности

Обычный случай это наличие нетривиального класса с методом `less`, используемом в операторе “меньше”, который, скорее всего объявлен вне класса.

```
1 struct S {
2     string first_name, last_name;
3     bool less(const S& rhs) const; // для(<)
4 };
```

Задача сделать для этого класса сравнение на равенство. Худшее что можно сделать это определить равенство механически.

```
1 bool operator==(const S& lhs, const S& rhs) {
2     return !lhs.less(rhs) && !rhs.less(lhs);
3 }
```

Гораздо лучше дописать в класс метод `equals` и определить в его терминах, ну или в случае с открытым содержимым просто сделать оператор.

```
1 bool operator==(const S& lhs, const S& rhs) {
2     return (lhs.first_name == rhs.first_name) &&
3            (lhs.last_name == rhs.last_name);
4 }
```

А вот с собственным хэшем это уже не слишком хорошо работает. Конечно можно напрячь фантазию.

```
1 result_type
2 operator()(argument_type const& s) const noexcept {
3     result_type const h1 = hash<string>{}(s.first_name);
4     result_type const h2 = hash<string>{}(s.last_name);
5     return h1 ^ (h2 << 1);
6 }
```

Сдвиг на единицу нужен для частого случая когда строки одинаковые. В данном случае это работает, но вообще, конечно, хотелось бы механического способа объединения хэшей. В стандарте его нет. На помощь может прийти `boost`, в частности `boost::hash_combine`.

```

1 result_type
2 operator()(argument_type const& s) const noexcept {
3     result_type const h1 = hash<string>{}(s.first_name);
4     result_type const h2 = hash<string>{}(s.last_name);
5     size_t seed = 0;
6     boost::hash_combine(seed, h1);
7     boost::hash_combine(seed, h2);
8     return seed;
9 }
```

Но тут, конечно, многие могут не захотеть затачивать boost в проект и их можно понять. Что же, рано или поздно это стандартизуют.

Кроме всего прочего, наличие в контейнере требования к такой функции, как `hash` открывает на уровне стандарта гораздо больше о внутренней структуре контейнера, чем оператор меньше. Относительно упорядоченных множеств и отображений здесь всегда употреблялось слово “дерево”, но с некоторой осторожностью. Хотя бы просто потому, что ничего в интерфейсе упорядоченных ассоциативных контейнеров не указывает на то, что там **должно быть** именно дерево. Может и дерево (и то не вполне ясно какое дерево). А может и не дерево вовсе.

А вот неупорядоченные ассоциативные контейнеры это совершенно точно разновидности хеш-таблиц. Тут всё очевидно и двух мнений быть не может. Даже неясно почему слово `hash` не внесли в название, как это сделали, например в Ruby.

Но хеш-таблица сама по себе это непрерывный массив корзин (buckets) из которых растут списки коллизий. А это означает, что она имеет много общего с последовательными контейнерами. В частности, что снова можно подумать о резервировании памяти.

```

1 unordered_map<int, Foo> mapNoReserve;
2 unordered_map<int, Foo> mapReserve;
3 mapReserve.reserve(1000);
4 for(int i = 0; i < 1000; ++i) {
5     mapNoReserve.insert({i, Foo()});
6     mapReserve.insert({i, Foo()});
7 }
8 // ..... control point
```

**Домашняя наработка:** найдите способ замерить полный размер неупорядоченного отображения в памяти. Проведите эксперимент, как в листинге выше. Какой размер в памяти имеют `mapReserve` и `mapNoReserve`

в контрольной точке?

## 5.10 Память своими руками

*Allocator relates to allocation, as vector to vexation*

– Andrei Alexandrescu

В тех языках, где память это ресурс (которых в современном мире уже, пожалуй, меньшинство), выделение и освобождение памяти это точки дополнительной интеллектуальной нагрузки на разработчика. Даже если оставить в стороне “проклятие меморилика”, выделение и освобождение памяти это всё равно потенциально дорого, а её неупорядоченное использование чревато странным поведением загадочных внутренних структур процессора, таких как кэши.

Стратегия выделения памяти по умолчанию, своего рода один размер, который подходит всем, это выделение с помощью менеджера памяти стандартной библиотеки (для C++ это глобальные операторы `new` и `delete`). Особенности этого механизма хорошо изучены, задокументированы в стандарте и нацелены на улучшение работы в среднем случае, когда памяти не выделяется ни мало ни много, а объекты аллокации равномерно большие и маленькие. Но что делать людям, чей софт не попадает под средний случай?

Стандартная библиотека C++ сразу была спроектирована таким образом, чтобы каждому контейнеру можно было предоставить собственный аллокатор, решающий вопросы выделения памяти. Например полный шаблон класса `vector` выглядит как:

```
1 template <class T, class A = std::allocator<T> > class vector;
```

Если вдуматься, это решение **странные**. STL проектировалась в 1992 году и не вполне ясно кому и зачем тогда могли понадобиться такие механизмы расширения.

Всё дело в позабытой ныне концепции `near` и `far pointers`. Дело в том, что в те годы было модно организовать память сегментами и указатель, показывающий за сегмент, мог иметь вдвое больший размер, чем глядящий внутрь сегмента. Производители аппаратуры предоставляли свои расширения, такие как `int * __huge px`, при этом стандарт языка об этом ничего не знал, понимая только обычные указатели, поэтому эти детали нужно было где-то спрятать.

Это повлияло на проектирование аллокаторов. По сути аллокаторы, исходно спроектированные Степановым, должны были знать всего две

вещи:

- Откуда взять память
- Как преобразовать её к  $T^*$  из платформенно-специфичного типа

Таким образом, исходно аллокаторы вообще не планировались как средство выделения памяти. Они планировались как тонкий слой адаптации к настоящему распределителю.

Вынесенные в эпиграф слова Александреску можно с сохранением образности перевести на русский: “Аллокатор относится к аллокации как барс к барсетке”.

Именно этот факт и создал впоследствии аллокаторам дурную славу: когда люди в итоге начали использовать аллокаторы для того, чтобы определять с их помощью стратегии выделения памяти, они столкнулись с неизбежными проблемами. И именно поэтому в C++ аллокаторы это наиболее часто пересматриваемый механизм: они существенно менялись в почти каждом стандарте, пока, наконец, в C++17 не стали почти совершенными.

Но об этом чуть позже.

### 5.10.1 Простые аллокаторы

Итак, вернёмся в 1998 год. Представьте, что у вас есть нестандартный системный распределитель `emalloc`, который распределяет память в расширенном адресном пространстве (EMS, если ктопомнит). И его хочется завести под вектора, списки, и прочие полезные контейнеры.

Простой аллокатор, который всего лишь знает как сбегать к распределителю и принести память, должен выставить использующему его контейнеру две соновные функции аллокаторного интерфейса: `allocate` и `deallocate`

```

1 template<typename T> struct ealloc {
2     typedef T value_type;
3     typedef T* pointer;
4     pointer allocate (size_t n) {
5         return static_cast<pointer>(emalloc(n * sizeof(T)));
6     }
7     void deallocate(pointer p, size_t n) { efree(p); }
8 };

```

Выше показан простой код, с которого вы, вероятно, начали бы. Кажется, что теперь получившийся адаптер можно скормить, например, вектору

```
1 template <typename T> using s_vector = vector<T, ealloc<T>>;
```

Не так быстро. Есть две проблемы и обе стали понятны сообществу как раз где-то к 98-му году.

## 1. Взаимозаменяемость аллокаторов

Можно проиллюстрировать так:

```
1 vector<int, ealloc<int>> v1, v2;
2 // a lot of code
3 v1 = v2; // what is happening here?
```

Если аллокатора в каком-то смысле похож на приведённый выше, то никаких проблем, привавание и есть присваивание. Но если аллокатор это объект с состоянием, как должно изменяться состояние аллокатора при копировании контейнера?

## 2. Приведение аллокаторов

Возмём для примера `list<T, ealloc<T>>`. Внутри себя список будет создавать не `T`, а `_list_node<T>`. То есть ему нужно иметь возможность как-то получить для внутреннего использования тип `dealloc<_list_node<T>>`, не имеющий с переданным ему `dealloc<T>` ничего общего.

Первому вопросу было суждено стать ключевым. Второй изначально казался попроще.

Стандарт 98 года решал первый вопрос просто: все конкретные экземпляры любого типа аллокаторов должны были быть эквивалентны. То есть единственная разумная запись операторов сравнения (и вы как программист должны её предоставить) для приведённого выше `dealloc` это следующий код.

```
1 template <typename T, typename U>
2 bool operator== (const ealloc<T>&, const ealloc<U>&) {
3     return true;
4 }
5
6 template <typename T, typename U>
```

```

7  bool operator!= (const ealloc<T>&, const ealloc<U>&) {
8      return false;
9  }

```

Для решения второй проблемы, внутри каждого аллокатора был предусмотрен (опять-таки предоставляемый программистом) шаблонный переходник `rebind`

```

1  template<typename T> struct ealloc {
2      // .... all the same ....
3      template<typename U> ealloc(const ealloc<U>&) {}
4      template<typename U>
5          struct rebind { typedef ealloc<U> other; };
6  };

```

Чтобы ещё немножко усложнить жизнь программистам, от вас также требовалось несколько чисто ритуальных движений: предоставить функции `construct` и `destroy`, выполняющие функции размещающего `new` и явного вызова деструктора (причём в реальности у вас не было вариантов написать их существенно иначе, чем показано ниже). Кроме того, от вас требовалась функция `max_size` для ограничения общего размера (которую тоже не было возможности сделать особо умной). А также куча синонимов вложенных типов для `reference`, `const_pointer` и всего такого.

```

1  template<typename T> struct ealloc {
2      // .... all the same ....
3      typedef const T* const_pointer;
4      typedef T& reference;
5      typedef const T& const_reference;
6      typedef size_t size_type;
7      typedef ptrdiff_t difference_type;
8
9      void construct(pointer p, const T& t) { new(p) T(t); }
10     void destroy(pointer p) { p->~T(); }
11     size_type max_size() const {
12         return numeric_limits<size_type>::max() / sizeof(T);
13     }
14 };

```

Все эти вещи были вынесены в traits в C++11 и объявлены устаревшими в C++14.

Вот теперь, пожалуй, всё. Такой аллокатор уже может использоваться в коде на C++98.

И тут, разумеется, встаёт главный вопрос: **для чего** он может использоваться. Хорошо, допустим предназначение `ealloc` было описано выше. Но что если хочется чего-то сложнее, чем просто нужную память подтаскивать и ненужную оттаскивать?

Первое, что приходит на ум это логгирующий аллокатор. Возможна также некая реализация защиты стека. Но что если хочется иметь заранее выделенный пул, где выделялись бы более мелкие объекты? Или, например сделать коалескинг при работе с этим пулом?

Короткий ответ: нет. Это также был ответ на большинство вопросов вида “а можно ли сделать ЭТО с аллокатором” и в 98 году он почти не зависел от того, чем было ЭТО.

Люди выкручивались как могли. Например собственный аллокатор, реализованный в библиотеке Intel Threading Building Blocks и серьёзно улучшающий производительность на многопоточных приложениях, работает по сути с собственной глобальной ареной, которая является частью рантайма этой библиотеки, также, как арене для `malloc` является частью рантайма `glibc`. Но это, конечно, ой.

В техническом развитии к 2003 году, в стандарте кристаллизовались два фундаментальных запрета, которые Мередит назвал “weasel words”. В точной формулировке они выглядели так:

- All instances of a given allocator type are required to be interchangeable and always compare equal to each other.
- The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `std::size_t`, and `std::ptrdiff_t`, respectively

В простом переводе эти фразы запрещали любые аллокаторы обладающие состоянием (например поддерживающие пул объектов) и любые попытки вернуть из аллокатора что-то кроме простого указателя (например умный указатель).

Это было настолько ограничивающим и мрачным, что такая крупная компания, как Блумберг, вошла в комитет по стандартизации только для того, чтобы найти способ это вычеркнуть.

### 5.10.2 Характеристики и состояние аллокаторов

Разумеется, когда к делу подключается тяжёлая артиллериya, многие вопросы решаются сами собой. В 2011 году ожидаемые многими изменения произошли. Если конкретнее:

- Аллокаторам разрешили не всегда быть равными, то есть разрешили обладать состоянием. Начиная с C++11 равенство аллокаторов означает, что один может освободить то, что выделил другой.
- Были ликвидированы ограничения для зависимого типа `pointer`. Теперь это может быть не просто `T*`, а практически всё что угодно, с некоторыми ограничениями.
- Был введён класс `allocator_traits` куда были собраны все редко переопределяемые вещи в качестве разумных умолчаний: `value_type`, `pointer`, `const_pointer` и так далее.
- Был введён класс `pointer_traits` для определения характеристик указателей.

Интересно, что проблема приведения указателя с помощью `rebind` при этом стало сложнее. Теперь из аллокатора типа `T` нужно иметь возможность получить не только аллокатор для типа `U`, но и характеристики из характеристик для `T`. Начиная с C++11 шаблон `allocator_traits` содержит зависимые шаблоны `rebind_alloc` и `rebind_traits`.

Определение для собственного класса тривиально

```

1 namespace std {
2     template <> struct allocator_traits<s_alloc> {
3         // implement all of traits
4     };
5 }
```

Но характеристик слишком много и такое переопределение обычно не нужно, так как содержит слишком много лишнего технически-бездумного кода.

На самом деле для того, чтобы изменить характеристику, достаточно переопределить соответствующий зависимый тип прямо в своём классе аллокатора.

Это достигается через SFINAE. Следующий код это выдержка из `libstdc++` с несколько менее уродливыми именами, чем там и правда есть.

```

1 template <typename Alloc> struct allocator_traits {
2 private:
3     using value_type = typename Alloc::value_type;
4     template <typename T> using ptrT = typename T::pointer;
5 public:
6     // either Alloc::pointer if defined
7     // or Alloc::value_type* if no Alloc::pointer
8     using pointer = detected_or_t<value_type*, ptrT, Alloc>;

```

Предполагается, что шаблонные параметры определителя `detected_or_t` это параметр по умолчанию, шаблон с произвольным числом аргументов и все эти аргументы. Примерно так:

```

1 template<typename Def, template<typename...> class Op,
2         typename... Args>

```

Он возвращает `Op<Args...>`, если он определён в sfinae-смысле или `Def` если нет.

**Вопрос к студентам:** как бы вы написали шаблон `detected_or_t`?

Аналогичным способом заведены и все остальные члены структур `allocator_traits` и `pointer_traits`.

Разумеется, характеристики аллокатора также содержат функции `allocate` и `deallocate`, а также `construct` и `destroy`.

Интересно тут следующее: функция `allocate` должна работать в терминах `pointer`, который может быть почти чем угодно, включая умный указатель или итератор. Но это явно не так для `construct`. Эта функция должна делать размещающий `new`.

```

1 template <typename Alloc> struct allocator_traits {
2     // .... everything else ....
3     static pointer allocate(Alloc &a, size_type n) {
4         return a.allocate(n);
5     }
6
7     template <typename T, typename ... Args>
8     static void construct(Alloc &a, T *p, Args&& ... args) {
9         a.construct(p, forward<Args>(args)...) ||

```

```

10     new (static_cast<void*>(p)) T(forward<Args>(args)...);
11 }
12 };

```

### 5.10.3 Пример: кэшированное освобождение

Идея free list аллокатора: освобождать блоки не в глобальный аллокатор а во free list

```
1 list<int, freelist_alloc<int>> l(v.begin(), v.end());
```

Следующие две строчки могут быть дорогостоящими если они действительно освобождают память

```

1 l.remove(2);
2 l.remove(6);

```

Но дело как раз в том, что никакого настоящего освобождения памяти тут не происходит. Два блока размером с `int` удаляются из списка и помещаются во freelist. Далее следующие две операции:

```

1 l.insert(l.begin(), -1);
2 l.insert(l.begin(), -3);

```

На самом деле ничего не выделяют, а всего лишь берут два блока из freelist, заполняют и возвращают в список

```

1 template <typename T>
2 class freelist_alloc {
3     using ST = aligned_storage_t<sizeof(T), alignof(T)>;
4     union node {
5         node* next;
6         ST storage;
7     };
8
9     node* list = nullptr;
10    // .... etc ....

```

Список устроен не на структуре, а на объединении, так как элемент, возвращаемый в список содержит только указатель для поддержания структуры списка, а при взятии из списка – только данные.

```
1 T* allocate(size_type n) {
```

```

2   if (n == 1) {
3     auto ptr = list;
4     if (ptr)
5       list = list->next; // from freelist
6     else
7       ptr = new node; // new node
8     return reinterpret_cast<T*>(ptr);
9   }
10
11  // taking new memory
12  return static_cast<T*>(::operator new(n * sizeof(T)));
13 }
```

Здесь показано как происходит выделение. Парная ей функция dealлокации выглядит тоже понятно

```

1 void deallocate(T* ptr, size_type n) noexcept {
2   if (n == 1) {
3     auto node_ptr = reinterpret_cast<node*>(ptr);
4     node_ptr->next = list;
5     list = node_ptr;
6     return;
7   }
8   ::operator delete(ptr);
9 }
```

Интересно также посмотреть на присваивание и перемещение. Присваивание не должно делать ничего особенного, поскольку присваивание объекта с таким аллокатором означает просто копирование состояния объекта, а оба аллокатора и слева и справа остаются теми же. А вот перемещение нетривиально, потому что по старому месту весь запасённый там freelist нужно освободить, заменив на свой.

```

1 freelist_alloc& operator= (const freelist_alloc&) noexcept {
2   return *this;
3 }
4
5 freelist_alloc& operator= (freelist_alloc&& other) noexcept {
6   if (this == &other)
7     return *this;
8   clear();
9   list = other.list;
10  other.list = nullptr;
```

```

11     return *this;
12 }
```

Таким образом, удивительно, но оператор присваивания и перемещение действуют очень по разному, при этом перемещение аллокатора **сложнее** копирования.

Если перечислять все случаи, как он могут действовать то аллокаторы могут: либо разделять некий ресурс (и тогда они должны пропагировать свой ресурс при копировании) либо каждый экземпляр аллокатора содержит локальный контекст и тогда не копируется (хотя при этом всё же может перемещаться, но это не точно).

И действительно, рассмотрим freelist.

**Вопрос к студентам:** что будет, если для freelist будет пропагация при копировании?

Как именно аллокатор будет вести себя на копировании или перемещении определяется начиная с C++11 следующими typedefs (ниже указаны также значения по умолчанию).

```

1 using propagate_on_container_copy_assignment = false_type;
2 using propagate_on_container_move_assignment = false_type;
3 using propagate_on_container_swap           = false_type;
4 using is_always_equal                     = false_type;
```

Некоторые считают, что они довольно уродливы. Да, конечно. Я полагаю, это сделано специально. Далее в этих лекциях немногочисленные ссылки на них будут обозначать сокращение по первым буквам: РОСА, РОСМА, РОСС, ИАЕ.

Переопределяя любой из них в `true_type`, вы, скорее всего, знаете, что делаете. Практически важный частный случай **локального** (ещё говорят arena-based) аллокатора.

#### 5.10.4 Локальные аллокаторы и их арены

Локальный аллокатор, как следует из его названия, привязан к конкретной локации. Его состояние это состояние распределение в этой конкретной строке, векторе, списке и т.д. При копировании локации, новая локация получает новый аллокатор с новым состоянием, после чего туда заводятся все существовавшие в старой локации элементы без каких-либо попыток пропагировать состояние собственно аллокатора.

Типичный пример полезного использования локального аллокатора – это вектор, оптимизированный для использования с небольшим количеством элементов.

```
1 template <class T, size_t BufSize = 200>
2 using SmallVector = vector<T, short_alloc<T, BufSize, alignof(T
)">>>;
```

Такой вектор располагается на стеке пока в нём меньше чем BufSize байт и пересоцируется в кучу, когда места на стеке не хватает

Аллокатор `short_alloc`, рассматриваемый в этом разделе, предложен Говардом Хинантом [49].

Ареной называется класс, управляющий локальным ресурсом

```
1 template <size_t N, size_t alignment = alignof(max_align_t)>
2 class arena {
3     char buf_[N] alignas(alignment);
4     char* ptr_;
5 public:
6     arena() noexcept : ptr_(buf_) {}
7     arena(const arena&) = delete;
8     arena& operator=(const arena&) = delete;
9     template <size_t ReqAlign> char* allocate(size_t n);
10    void deallocate(char* p, size_t n) noexcept;
```

Но при этом арена не является аллокатором и не реализует интерфейс аллокатора. Что не мешает ей иметь внутри шаблонный метод `allocate`

```
1 template <size_t N, size_t alignment>
2 template <size_t ReqAlign>
3 char *arena<N, alignment>::allocate(size_t n) {
4     auto const aligned_n = align_up(n);
5     auto bsz = static_cast<decltype(aligned_n)>(buf_ + N - ptr_);
6     if (bsz < aligned_n)
7         return static_cast<char*>(::operator new(n));
8     char* tmp = ptr_;
9     ptr_ += aligned_n;
10    return tmp;
11 }
```

При аллокации, как и в случае `freelist` есть разветвка. Используется либо временный буфер, либо глобальный `new`.

Для того, чтобы связать конкретную арену с интерфейсом аллокатора используется отдельный класс `short_alloc`.

```

1 template <class T, size_t N, size_t A = alignof(max_align_t)>
2 class short_alloc {
3     arena<N, A>& a_;
4 public:
5     short_alloc(arena_type& a) noexcept : a_(a) {}
6     T* allocate(size_t n) {
7         char *res = a_.allocate<alignof(T)>(n * sizeof(T));
8         return reinterpret_cast<T*>(res);
9     }

```

Единственное неудобство вектора с таким аллокатором: нужно сначала где-то (например на стеке) выделить арену, а уже потом связать её с экземпляром контейнера.

```

1 SmallVector<int>::allocator_type::arena_type a;
2 SmallVector<int> v{a};

```

Но после этого небольшого трюка, использование не сложнее, чем использование обычного вектора.

### 5.10.5 Проблемы области распространения

Коль скоро у аллокаторов теперь есть состояние, одного их типа более недостаточно, важным становится конкретный экземпляр аллокатора. Это не всегда так, например в случае с `short_alloc` имел значение экземпляр арены. Но очень часто это так.

Пусть существует какой-нибудь аллокатор `CustomAlloc`. В коде, приведённом ниже, три его разных экземпляра используются как аллокаторы для трёх разных (условно) строк.

```

1 using CustomStr =
2     string <char, char_traits<char>, CustomAlloc<char>>;
3 CustomAlloc<char> alloc1(SYSTEM), alloc2(LOCAL), alloc3;
4 CustomStr x1(alloc1), x2(alloc2), x3(alloc3);

```

Теперь в программе есть три строки, память которых управляетя тремя разными аллокаторами одного и того же типа. Это порождает странные и удивительные проблемы.

Например можно рассмотреть кажущийся вполне разумным сценарий, когда они все помещаются в вектор.

```

1 vector<CustomStr> vec;
2 vec.push_back(x1);
3 vec.push_back(x2);
4 vec.reserve(4);

```

Результат проиллюстрирован на (рис. 5.27)

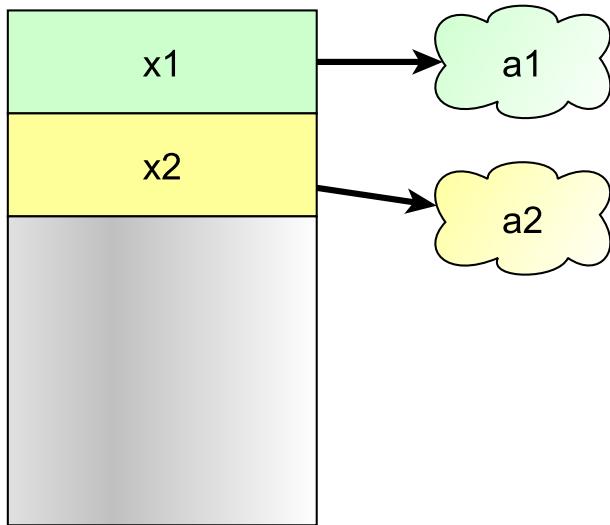


Рис. 5.27: Помещение в вектор двух строк с локальными аллокаторами

Пока что всё выглядит отлично: в одном векторе живут два элемента, каждый на своей позиции и каждый со своим локальным аллокатором.

Увы, следующий крайне простой трюк усложняет вещи совершенно непропорционально.

```
1 vec.insert(vec.begin(), x3);
```

Что происходит при выполнении этой строчки? Сначала все элементы должны быть сдвинуты вниз. Первым сдвигается  $x_2$  и заводит на новом месте свой аллокатор. За ним вниз ползёт  $x_1$ , но в ячейке 2 уже есть аллокатор от  $x_2$  и он там остаётся. Далее  $x_3$  помещается в ячейку 1, где работает аллокатор для 1.

На итоговой картинке (рис. 5.28) получается довольно устрашающая картина.

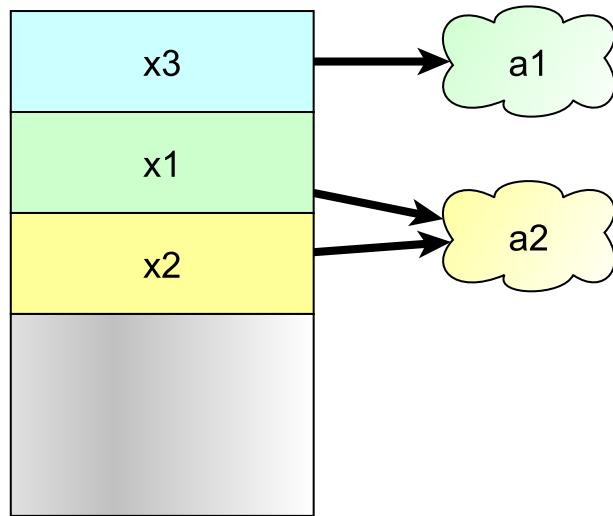


Рис. 5.28: Иллюстрация проблем локальных аллокаторов

Почему так получилось? До некоторой степени мы вообще не хотели бы чтобы строки тащили в ячейки вектора собственные аллокаторы: будет вполне достаточно если они воспользуются аллокатором вектора.

Но для этого вектор должен уметь сообщать каждой помещаемой в него строке, что у него есть свой аллокатор которым она должна пользоваться. Говорят, что у вектора должен быть распространяемый (scoped) аллокатор. Тут есть небольшая омонимия, так как на английском языке `scope` также означает область видимости.

Для этих целей, служит шаблон `scoped_allocator_adapter` в стандарте C++11.

Ниже приведён пример того, как это работает для случая близкого к рассмотренному.

```

1 namespace cust
2 {
3     template <typename T> CustomAlloc;
4     template <typename T> using alloc =
5         ::std::scoped_allocator_adapter<CustomAlloc<T>>;
6     template <typename T> using vector = ::std::vector<T, alloc<T
7         >>;
8     template <typename T> using string =
9         ::std::basic_string<char, char_traits<char>, alloc<char>>;
9 }
```

```

10
11 cust::vector<cust::string> vs(&alloc1); // propagated ok

```

Теперь аллокаторы правильно пропагируются, но поглядите на этот код ещё раз. Он чудовищен. Он не может нравиться эстетически, к нему можно только привыкнуть.

Даже для обычных аллокаторов тотальное загрязнение кода вирусными шаблонами заставляет глаза кровоточить даже у опытных разработчиков.

```

1 map <int, float, less<int>, s_alloc<pair<const int, float>>> m;
2 basic_string<char, char_traits<char>, s_alloc<char>> s;

```

Но самый убийственный аргумент против аллокаторов, обладающих состоянием это аргумент от сложности написания контейнера, корректно поддерживающего произвольные аллокаторы.

Вот достаточно условное перемещающее присваивание контейнера такого рода. Это песвдокод. В реальности проверки РОСМА и ИАЕ будут выглядеть как объяснялось выше (см. 5.10.3).

```

1 container& operator= (constainer &&rhs) {
2     if (alloc_traits::POCMA) {
3         clear_and_deallocate_memory();
4         alloc_ = move(rhs.alloc_);
5         impl_ = move(rhs.impl_);
6     }
7     else if (alloc_traits::IAE || alloc_ == rhs.alloc_) {
8         clear_and_deallocate_memory();
9         impl_ = move(rhs.impl_);
10    }
11    else {
12        this->assign(move_iterator(rhs.begin()),
13                      move_iterator(rhs.end()));
14    }
15    return *this
16 }

```

Здесь написано следующее:

1. Если аллокатор пропагируется на перемещающем присваивании, то достаточно обычного перемещения контейнера и перемещения аллокатора на новое место.

2. Иначе, если аллокатор сравним на равенство и оказался равным аллокатору справа, то достаточно перемещения контейнера
3. Иначе мы должны провести дорогое копирование содержимого

По этой причине написание поддерживающего аллокаторы контейнера в реалиях C++11 это крайне интересная тема, но здесь она рассматриваться не будет, потому что, с точки зрения автора, гуманнее включённый миксер в глаза засунуть.

Причём вроде бы на пути к этому ужасу, всё было сделано правильно и логично. Это значит, что неправильный поворот был пройден довольно рано.

Вполне возможно, следует откатится к самому раннему решению: а почему вообще все исходно начали считать, что тип аллокатора должен быть частью типа контейнера?

Допустим аллокатор вообще не должен быть частью типа контейнера. В этом случае он должен быть всегда scoped и никогда не должен копироваться и перемещаться. И это приводит к идее, полноценно реализованной в C++17: идее полиморфного аллокатора.

### 5.10.6 Полиморфные аллокаторы

Когда в самом начале главы (см. 5.10.1) рассматривался возможный дизайн наивного аллокатора, он был спроектирован довольно простым образом

```

1 template<typename T>
2 struct memory_resource {
3     T* allocate (size_t n);
4     void deallocate(T* p, size_t n);
5 };

```

Пока что не следует обращать внимание на то, что я изменил название, скоро это получит своё объяснение.

Что будет, если из этого кода удалить типы?

```

1 struct memory_resource {
2     virtual void* allocate (size_t n,
3         size_t align = alignof(std::max_align_t)) = 0;
4     virtual void deallocate(void* p, size_t n) = 0;

```

```

5     virtual bool is_equal(const memory_resource&) const = 0;
6 };

```

Время показало, что также очень полезной функцией для аллокаторов является проверка на равенство, поэтому она здесь добавлена.

Но в целом проектирование тут не слишком удачное, потому что параметр по умолчанию в виртуальной функции это очень нехорошо. Он связывается статически и породит массу проблем, описанных ранее (см. 3.6.4).

Выходом тут является идиома NVI (см. 3.7.2)

```

1 struct memory_resource {
2     void* allocate(size_t n, size_t align = alignof(max_align_t))
3         ;
4     void deallocate(void* p, size_t n);
5     bool is_equal(const memory_resource&) const noexcept;
6     protected:
7         virtual void* do_allocate(size_t n, size_t align) = 0;
8         virtual void do_deallocate(void* p, size_t n) = 0;
9         virtual bool do_is_equal(const memory_resource&) const
10             noexcept = 0;
11 };

```

Получившийся класс не является аллокатором, потому что теперь он не удовлетворяет требованиям 2011-го года к интерфейсу аллокатора. Но вся прелесть в том, что он и **не называется** аллокатором. Это полиморфный ресурс в памяти, от которого теперь можно наследовать всякие интересные ресурсы.

В стандарте прописаны следующие стандартные наследники

- `null_memory_resource` – самый интересный ресурс, всегда `nullptr`
- `new_delete_resource` – стандартный ресурс с `new/delete`
- `synchronize_pool_resource` – мультиплул с многопоточной синхронизацией
- `unsynchronize_pool_resource` – быстрый мультиплул без синхронизации
- `monotonic_buffer_resource` – монотонное выделение

Тут встречаются два новых термина – мультипул (multipool) и монотонное (monotonic) выделение. Это две стратегии работы с памятью, настолько себя зарекомендовавшие, что их предложили в стандарт. Монотонный ресурс это ресурс, который монотонно выделяет память внутри некоего заранее выделенного буфера. Память не освобождается, в конце работы прибивается сам буфер (стоит памяти при частых алокациях). Мультипул ресурс это несколько связанных пулов, в которые выделяется и освобождается память. Пулы преаллоцируются и при нехватке, выделяется больший и больший (ускоряет работу при алокациях/деаллокациях)

Теперь, при наличии ресурса, осталось только сделать к нему адаптер, который является честным C++11 алокатором.

```

1 template<typename T> struct polymorphic_allocator {
2     polymorphic_allocator();
3     polymorphic_allocator(memory_resource *mr);
4     T* allocate (size_t n);
5     void deallocate(T* p, size_t n);
6     // .... etc ....
7 private:
8     memory_resource* mr_;
9 };

```

Выше приведён псевдокод, потому что настоящий `polymorphic_allocator` должен быть всегда scoped, то есть наследовать от `scoped_allocator_adaptor`.

**Вопрос к студентам:** как бы вы реализовали копирующее присваивание для `polymorphic_allocator`?

Начиная с C++17, вся стандартная библиотека с полиморфной алокацией включена в пространство имён `std::pmr`.

```

1 namespace pmr {
2     template <class T> using vector =
3         ::std::vector<T, std::pmr::polymorphic_allocator<T>>;
4     // .... all other containers ....
5 }

```

Для удобства туда же включены классы полиморфного алокатора, основные ресурсы в памяти и прочее.

Следующий код это пример ещё одного вектора на стеке, на этот раз в реалиях `pmr`.

```

1 constexpr size_t sz = 1000 * sizeof(double);
2 char buffer[sz] alignas(double);
3 pmr::monotonic_buffer_resource alloc(buffer, sz);
4 double start = 0.0;
5 pmr::vector<double> v1(&alloc);
6 generate_n(back_inserter(v1), 100,
7   [start] () mutable { return (start += 1.1); });

```

Здесь прекрасно всё, но особенно то, что вообще нет аллокаций в динамическую память.

Иногда стандартных ресурсов недостаточно и хочется написать свой ресурс, который инкапсулировал бы свою уникальную стратегию распределения памяти.

### 5.10.7 Иерархия ресурсов в памяти

Для примера можно разработать тестовый ресурс. Он проверяет что аллокация соответствует dealлокации и пытается отслеживать утечки памяти.

```

1 struct test_resource : public pmr::memory_resource {
2     // interface here
3 private:
4     struct allocation_rec {
5         void *ptr_;
6         size_t nbytes_, nalign_;
7     };
8     pmr::memory_resource *parent_;
9     pmr::vector<allocation_rec> blocks_;
10 };

```

Переопределение do\_allocate

```

1 void *test_resource::do_allocate(size_t bytes, size_t align) {
2     void *ret = parent_->allocate(bytes, align);
3     blocks_.emplace_back(ret, bytes, align);
4     return ret;
5 }

```

Видно, что тестовый ресурс просто сцепляется с тем, над которым он живёт. Это обычная идея: теперь можно складывать `memory_resources` в иерархические стопки.

Но любая иерархия неизбежно порождает вопрос, кто в этой иерархии находится сверху.

```
1 static test_resource newdefault{pmr::new_delete_resource()};
2 pmr::set_default_resource(&newdefault);
```

Для установки ресурса по умолчанию для pmr контейнеров, в стандарте был оставлен хук `pmr::set_default_resource` и это худшее, что там было оставлено. Простейший способ нарваться на проблемы тут, это, например, забыть слово `static`.

**Вопрос к студентам:** кстати, а что будет, если тут забыть слово `static`?

Итак, начиная с 2017 года, собственных аллокаторов писать вообще не надо, так как полиморфного аллокатора хватит на всех, а собственные ресурсы в памяти писать действительно просто. Что насчёт собственных контейнеров? Конечно их тоже стало писать удивительно просто.

### 5.10.8 Собственные контейнеры, пользующиеся аллокаторами

Для примера подобного рода контейнера можно взять что-то, чего нет в стандарте. В докладе господина Халперна на CppCon [42] рассматривался альтернативный дизайн односвязного списка, поддерживающего быстрый `size` ценой медленного `splice` и быструю вставку как в хвост, так и в голову. Принципиальное устройство такого списка показано на (рис. 5.29).

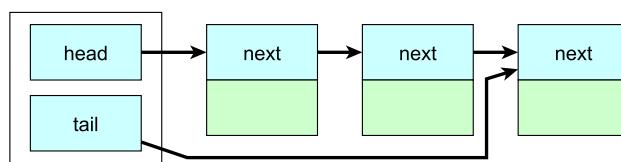


Рис. 5.29: Альтернативный односвязный список

Как может быть устроен узел для такого контейнера? Очевидная для программиста с первого взгляда схема на листинге ниже имеет один недостаток.

```
1 template <typename Tp> struct node;
2 template <typename Tp> struct node_base {
3     node<Tp> *next_ = nullptr;
```

```

4     node_base(const node_base&) = delete;
5     node_base& operator=(const node_base&) = delete;
6 };
7 template <typename Tp> struct node : node_base<Tp> {
8     Tp value_;
9 };

```

Недостаток тут в том, что сложно придумать разумное значение по умолчанию для `value_`. Оно будет инициализировано при конструировании но чем и зачем – неясно.

Выход в распространённом трюке, использующем union: начиная с 2011 года объединения в C++ могут объединять нетривиальные объекты с учётом, что пользователь сам заботится об их памяти.

```

1 template <typename Tp> struct node : node_base<Tp> {
2     union { Tp value_; };
3 };

```

Это улучшение позволяет внести немного неочевидной job security в код и, одновременно, сэкономить пару тактов времени исполнения (но последнее не точно). Какой же C++ разработчик удержится.

Скелет для итогового класса списка с некоторыми важными частями внутри приведён ниже.

```

1 template <typename T> struct slist {
2     using value_type = T;
3     using allocator_type = pmr::polymorphic_allocator<byte>;
4     // .... etc ....
5 public:
6     slist(allocator_type a = {}) :
7         head_{}, ptail_{&head_}, size_{0}, alloc_{a} {};
8     allocator_type get_allocator() const { return alloc_; }
9     // .... etc ....
10 private:
11     node_base head_;
12     node_base *ptail_;
13     size_t size_;
14     allocator_type alloc_;
15 };

```

Тот факт, что список способен пользоваться произвольным полиморфным аллокатором, здесь считывается из двух обязательных пунктов:

1. Наличия зависимого типа `allocator_type`
2. Наличия метода `get_allocator`

Тип `polymorphic_allocator<byte>` здесь это как бы общий базовый класс для всех аллокаторов. В принципе содержимое треугольных скобок у него не слишком важно: при наличии ребинда, какая разница для какого он типа. Это всего лишь тяжёлое легаси интерфейса 1998-го года.

Удивительное наблюдение тут состоит в том, что все составляющие хороших и правильных аллокаторов уже были известны в 98-м году, но люди стремились к шаблонной сложности и получили её. Урок состоит в том, что хороший программист стремится не к сложности, а к простоте.

Ключевым методом контейнера `slist`, разумеется, является `emplace`, поскольку именно в его терминах будет реализована вся остальная вставка.

Скелет правильного создания

```

1 template <typename ... Args> iterator
2 emplace (iterator i, Args&& ... args) {
3     void *mem = alloc_.resource()->allocate(sizeof(node), alignof
4         (node));
5     node *ret = static_cast<node*>(mem);
6     ret->next_ = i.prev->next_;
7     alloc_.construct(addressof(ret->value_), forward<Args>(args)
8         ...);
9     i.prev->next_ = ret;
10    // .... some corner cases ....
11 }
```

**Вопрос к студентам:** кто видит проблему в этом коде?

### 5.10.9 Сводная таблица

Аллокаторы это такой своеобразный и так часто меняющийся механизм стандартной библиотеки, что в нём ничего не стоит запутаться.

Будут ли они меняться дальше или придуманные в 2017 году полиморфные аллокаторы совершены? Скорее всего, ещё несколько итераций изменения неизбежны. Простейший аргумент это аргумент от стоимости поддержки.

Представьте обычную структуру, состоящую из целого числа и вектора.

```

1 struct S {
2     int n;
3     vector<int> v;
4 };

```

Сейчас, так как она написана, эта структура имеет по умолчанию всё необходимое: копирование, присваивание, всё остальное. Кроме поддержки аллокаторов. Сколько кода надо написать, чтобы завести в ней эту поддержку? Увы, даже сейчас это тонны кода. Но чисто интуитивно в этом случае (вектор плюс один элемент) поддержка не должна стоить ничего: вектор уже всё поддержал и завести ещё одно целое под этот зонтик – технический момент.

Для простоты отслеживания, изменения сложности простейших задач (в весьма условных попугаях) сведены в таблицу.

| Task              | C++98      | C++11  | C++17  |
|-------------------|------------|--------|--------|
| Использование     | MEDIUM     | MEDIUM | EASY   |
| Создание          | MEDIUM     | EASY   | EASY   |
| Создание stateful | IMPOSSIBLE | EASY   | EASY   |
| Создание scoped   | IMPOSSIBLE | MEDIUM | EASY   |
| Новый контейнер   | MEDIUM     | HARD   | MEDIUM |

Это завершает обсуждение аллокаторов.

## 5.11 Концепты

*Thoughts without content are empty,  
intuitions without concepts are blind*

– Immanuel Kant

Любую компьютерную программу читает (и иногда пишет) разработчик, который оперирует семантическим смыслом выражений языка программирования. При этом у разработчика есть очень большое количество корректных синтаксических форм для семантически эквивалентных конструкций.

Но ту же программу читает (обрабатывает) компилятор, который не понимает её смысла. Вместо этого компилятор механически упрощает выражения и трансформирует их, ориентируясь на их финальное представление в целевой архитектуре. Так выражению имеющему смысл сложения величин могут быть сопоставлены принципиально разные инструкции ассемблера в зависимости от того являются эти числа целыми или с плавающей точкой.

Именно тот факт, что смысл выражений высокогоуровневых языков часто совершенно не зависит от типов и конкретной интерпретации этих выражений, позволяет обобщенное программирование как таковое.

Почти классический пример обобщенного кода: функция для нахождения элемента в диапазоне (диапазон здесь может быть вектором, списком, деком, множеством, пользовательским типом, согласующимся с сокращенным синтаксисом `for`). Это (а также тип элемента) серьёзно влияет на конкретный генерированный код, но совсем не влияет на смысл следующей простой конструкции:

```
1 template<typename R, typename T> bool
2 in (R const& range, T const& value)
3 {
4     for (auto const& x : range)
5         if (x == value)
6             return true;
7     return false;
8 }
```

**Вопрос к студентам:** что будет при попытке использовать этот код неправильно, например “найти” число в векторе строк?

```

1 vector<string> v { "0", "1", "2" };
2 bool is_in = in (v, 0);

```

Очевидно ошибка компиляции. Главная проблема в том, что это будет **неприятная** ошибка компиляции:

```

In file included from /home/tilir/Applications/gcc-5.2/include
/c++/5.2.0/bits/stl_algobase.h:67:0,
                 from /home/tilir/Applications/gcc-5.2/include
/c++/5.2.0/vector:60,
                 from constr1.cc:1:
/home/tilir/Applications/gcc-5.2/include/c++/5.2.0/bits
/stl_iterator.h:820:5: note: candidate:
template<class _IteratorL, class _IteratorR, class _Container>
bool __gnu_cxx::operator==(const __gnu_cxx::__normal_iterator
<_IteratorL, _Container>&, const __gnu_cxx::__normal_iterator
<_IteratorR, _Container>&)
    operator==(const __normal_iterator
<_IteratorL, _Container>& __lhs,
^
/home/tilir/Applications/gcc-5.2/include/c++/5.2.0/bits
/stl_iterator.h:820:5: note:   template argument deduction
/substitution failed:
constr1.cc:10:11: note: ‘const std::__cxx11::basic_string<char>’
is not derived from ‘const __gnu_cxx::__normal_iterator
<_IteratorL, _Container>’
        if (x == value)

```

Это только крошечный кусочек из простины которую выдаст gcc-5.2

**Домашняя наработка:** попробуйте скомпилировать ещё более простой код:

```

1 class X {};
2 std::set<X> x;
3 x.insert(X{});

```

В своё время обобщенное программирование на C++ сильно споткнулось об это. Эти знаменитые простины ошибок, которые заставляли даже опытных разработчиков кричать и плакать как детей, увы, неизбежны. Шаблоны добавляют лишний шаг к компиляции программы: сначала

должны быть инстанцированы типы, потом скомпилирован код. В случае если инстанцирование создало большой стек, будет много ошибок, без рациональной возможности сократить их, если нет механизмов ограничить шаблонный полиморфизм и заблокировать собственно инстанцирование. Между прочим, безо всякого шаблонного инстанцирования точно таких же безумных ошибок можно добиться макросами.

```
1 #include __FILE__  
2 p;
```

Идея здесь та же – макроподстановка никак не ограничена. Но макроподстановка не Тьюринг полна, поэтому в макросах это менее актуально (там обычно просто нет достаточно большого стека подстановки чтобы развернуться). Хотя если кто-то хоть раз пробовал boost preprocessor, он знает боль. А в шаблонах рекурсивные вызовы – обычное дело. И тем важнее добиться того, чтобы неправильные ветки отсекались как можно раньше.

Итак ключ к обоснованию нововведений в языке и главная тема этой лекции: **ограничение шаблонного полиморфизма**.

Ограничить шаблонный полиморфизм – также хорошая идея для разделения типов: иногда хотелось бы такой обобщенный тип, который ведет себя по разному когда инстанцирован чем-то вроде целого или чем-то вроде плавающего числа (может быть даже имеет разные методы).

Итак, ограничений и контроля полиморфизма в языке явно не хватало. И, конечно, все всегда понимали, что с этим надо что-то делать. За много лет коммюнити наработало некий опыт.

### 5.11.1 Добрые старые способы контроля полиморфизма

В этом разделе будут рассмотрены методы, работающие для C++14. Поскольку это часто связано с написанием большого количества кода, лучше в качестве рабочего примера рассмотреть очень простую обобщенную функцию проверки значений двух разных типов на равенство.

```
1 template <typename T, typename U> bool  
2 check_eq (T &&lhs, U &&rhs) {  
3     return (lhs == rhs);  
4 }
```

Её также можно использовать правильно или ошибочно.

```

1 // no operator == here
2 struct noeql {
3     noeql (int x) {}
4 };
5
6 int main () {
7     check_eq (1, 1);      // ok
8     check_eq (1, 1.0);   // ok
9     check_eq (1, noeql(1)); // problems
10    return 0;
11 }
```

В случае ошибочного использования, ошибка компиляции выглядит очень простой:

```

eqcomp00.cc: In instantiation of
  ‘bool check(T&&, U&&) [with T = int; U = noeql]’:
eqcomp00.cc:13:20:   required from here
eqcomp00.cc:3:15: error: no match for
  ‘operator==’ (operand types are ‘int’ and ‘noeql’)
    return (lhs == rhs);
```

Но, тем не менее, хотелось бы вместо неё иметь нечто осмысленное.

Простейший способ, и, обычно, первый, который приходит в голову новичку, это влепить статическую проверку.

```

1 static_assert (is_equality_comparable<T, U>::value,
2                 "Comparable types required");
```

Для этого надо сначала понять, что такое `is_equality_comparable`. В стандарте C++14 ничего такого нет, но хороший программист без труда спляшет это ограничение на SFINAE в C++14.

```

1 template <typename T, typename U, typename = void>
2 struct is_equality_comparable : std::false_type {};
3
4 // I feel really C++ here
5 template <typename T, typename U>
6 struct is_equality_comparable <T, U,
7     decltype ((void)(std::declval<T>() == std::declval<U>()))
8 > : std::true_type {};
```

Стало ли лучше?

```
eqcomp00.cc: In instantiation of
‘bool check(T&&, U&&) [with T = int; U = noeq]’:
eqcomp00.cc:28:27:   required from here
eqcomp00.cc:16:3: error: static assertion failed:
    Comparable types required
    static_assert (is_equality_comparable<T, U>::value,
    ^
eqcomp00.cc:18:15: error: no match for
‘operator==’ (operand types are ‘int’ and ‘noeq’)
return (lhs == rhs);
```

Стало хуже. Мы получили static assertion failure и, **в добавок** к нему, также instantiation failure.

Проблема в том, что эти ошибки проявляются **поздно**. Компилятор уже начал инстанцирование шаблона и он уже прошел по этому пути достаточно далеко. Наличие или отсутствие failures далее ничего не значит. Таким образом, `static_assert` в языке нужен не для этого. Он не мешает компилятору разобрать функцию, а защищает от ошибок, которые вызваны несовпадением статических контрактов.

Гораздо интереснее использовать `enable_if`, который даёт возможность выбросить функцию из рассмотрения:

```
1 // Wow, I am even MORE C++ now
2 template <typename T, typename U,
3           typename = std::enable_if_t <
4             is_equality_comparable<T, U>::value>
5         >
6 bool check (T &&lhs, U &&rhs);
```

В `enable_if<false, T>` нет зависимого типа `enable_if<false, T>::type` и происходит классический случай SFINAE, см. (4.8.3).

Ошибка выглядит более вразумительно:

```
eqcomp01.cc:27:20: error:
no matching function for call to ‘check(int, noeq)’
check (1, noeq(1));

eqcomp01.cc:18:6: note: candidate:
```

```
template<class T, class U, class> bool check(T&&, U&&)
bool check (T &&lhs, U &&rhs);
^
eqcomp01.cc:18:6: note:
template argument deduction/substitution failed
```

Инстанцирования такой функции просто не происходит. Тем не менее, сложно не заметить, что синтаксис для вышивания на `enable_if` пере усложнен и требует для блокирования подстановки завязываться на третий шаблонный параметр, который в общем не при чём.

### 5.11.2 Простые ограничения

Введение в новый стандарт синтаксического сахара для таких конструкций было практически неизбежным.

```
1 template <typename T, typename U>
2 requires is_equality_comparable<T, U>::value
3 bool check (T &&lhs, U &&rhs);
```

Теперь правильное использование всё так же идёт без ошибок. В то же время неправильное использование выдает простую ошибку:

```
eqcomp01s.cc: In function ‘int main()’:
eqcomp01s.cc:27:20: error: cannot call function
  ‘bool check(T&&, U&&) [with T = int; U = noeq]’
    check (1, noeq(1));
^
eqcomp01s.cc:18:6: note:   constraints not satisfied
  bool check (T &&lhs, U &&rhs);
~~~~~
eqcomp01s.cc:18:6: note:
  ‘is_equality_comparable<T, U>::value’ evaluated to false
```

Главное в этой ошибке то, что она, как и в случае `enable_if`, выдается **рано**, то есть задолго до того, как компилятор реально полез инстанцировать что бы то ни было. Sutton сообщал об уменьшении времени компиляции хорошо ограниченного таким образом кода на десятки процентов.

Явная запись `requires` в таком контексте называется ограничением (constraint) и в данном случае используется для проверки ограничения функцию стандартной библиотеки.

Также это является *requires-clause* – собственно ограничением. Немного позже будут введены *requires-expressions* – то, что будет называться ограничивающими выражениями. Уже сейчас хочется предупредить, что их нельзя путать.

Несмотря на то, что в этом разделе ограничения вводятся как ограничения для обобщенного программирования, на самом деле в них нет ничего магического: это обычный C++ и они могут использоваться с обычными функциями.

```
1 void f ()  
2     requires sizeof(int) == 4  
3 {  
4     /* do something */  
5 }
```

И даже более того, нешаблонные функции могут быть ограничены охватывающими шаблонными параметрами.

```
1 template <typename T>  
2 class vector {  
3     // ...  
4     // we have copy ctor only if T copyable  
5     requires std::is_copyable<T>::value  
6     vector (const vector&);  
7 };
```

**Домашняя наработка:** попробуйте такое сделать на `enable_if` просто чтобы понять масштаб упрощения.

Констрайнты работают лениво: возможное нарушение размера целого будет проверено только в точке вызова этой функции. Нет вызова – нет проблем.

**Вопрос к студентам:** кстати об ограничениях на ограничения. А как вы думаете, виртуальные функции могут быть ограничены подобным образом?

### 5.11.3 Пример – перегрузка конструкторов

Во многих случаях даже простых констрайнтов достаточно, чтобы чудесно упрощать код. Типичная задача – внутри обобщенного класса иметь отдельно конструктор из плавающих и отдельно из целочисленных типов (пример взят из доклада Роджера Орра на ACCU 2016).

Если решать эту задачу на бумажке, на ум приходит решение в старом добром стиле.

```

1  struct Foo {
2      enum {int_like, float_like} type_;
3
4      template <typename Int,
5                  typename = std::enable_if_t<
6                      std::is_integral<Int>::value>
7              >
8      Foo (Int x) : type_(int_like) {
9          std::cout << "int like: " << x;
10     }
11
12     template <typename Float,
13                 typename = std::enable_if_t<
14                     std::is_floating_point<Float>::value>
15                 >
16     Foo (Float x) : type_(float_like) {
17         std::cout << "float like: " << x;
18     }
19 };
20
21 int main () {
22     Foo (1);
23     Foo (5.0);
24     return 1;
25 }
```

Увы, у него есть небольшой недостаток – оно не скомпилируется.

```
ctorex01.cc:13:3: error:
‘template<class Float, class> Foo::Foo(Float)’
  Foo (Float x) : type_(float_like)
  ^~~~
```

```
ctorex01.cc:9:3: error: with
‘template<class Int, class> Foo::Foo(Int)’
  Foo (Int x) : type_(int_like)
```

Шаблонные функции просто не могут быть перегружены по имени шаблонного параметра. Впрочем, танки грязи не боятся и немногого времени в отладчике приводят к более интересному решению.

```
1 struct Foo {
2
3     enum {int_like, float_like} type_;
4
5     template <int> struct dummy { dummy(int) {} };
6
7     template <typename Int,
8             typename = std::enable_if_t<
9                 std::is_integral<Int>::value>
10            >
11     Foo (Int x, dummy<0> = 0) : type_(int_like)
12     {
13         std::cout << "int like: " << x << std::endl;
14     }
15
16     template <typename Float,
17             typename = std::enable_if_t<
18                 std::is_floating_point<Float>::value>
19            >
20     Foo (Float x, dummy<1> = 0) : type_(float_like)
21     {
22         std::cout << "float like: " << x << std::endl;
23     }
24 };
```

Теперь все хорошо, но опять ценой некоторой избыточности. В то же время даже с самыми простыми ограничениями та же задача решается как персик.

```
1 struct Foo {
2
3     enum {int_like, float_like} type_;
4
5     template <typename Int>
```

```

6   requires std::is_integral<Int>::value
7   Foo (Int x) : type_(int_like)
8   {
9     std::cout << "int like: " << x << std::endl;
10 }
11
12 template <typename Float>
13 requires std::is_floating_point<Float>::value
14 Foo (Float x) : type_(float_like)
15 {
16   std::cout << "float like: " << x << std::endl;
17 }
18
19 };

```

Со скромной точки зрения автора, даже если бы ограничения ввели только в виде простых ограничений, это серьёзно разгрузило бы метaprogramмирование во многих практически важных контекстах.

#### 5.11.4 Сложные ограничения

Простыми ограничениями можно написать три вида выражений: предикаты, конъюнкции предикатов и дизъюнкции предикатов.

**Вопрос к студентам:** пусть задан предикат:

```

1 template <typename T>
2 constexpr int somepred() {
3   return 14;
4 }
```

выполняются или нет указанные констрайнты для функции `check` если первый терм ложен?

```

1 template <typename T, typename U>
2 requires is_equality_comparable<T, U>::value &&
3   (somepred<T>() == 14) ||
4   (somepred<U>() == 42)
5 bool check (T &&lhs, U &&rhs);
```

К сожалению, это не всё, что может быть нужно от ограничений. Простой пример: как избавиться от всё ещё необходимого шаблонного

определения `is_equality_comparable`? Для этого может быть использовано сложное ограничение, использующее ограничивающие выражения (requires-expressions).

```

1 template <typename T, typename U>
2 requires requires(T t, U u) { t == u; }
3 bool check (T&& lhs, U&& rhs);

```

Ключевое слово `requires` упомянутое дважды делает именно это – открывает ограничение, после чего вместо предиката использует в нем ограничивающее выражение.

Ограничивающие выражения похожи на константно-выраженные (constexpr) функции. Но в них есть своя магия. Она заключается в том, что сложные ограничения никогда не вычисляются, в этом они похожи на выражения-параметры decltype. Для них только проверяются типы. Это создаёт громадную разницу между двумя формами ограничений

```

1 // (1) constexpr function somepred ct evaluated
2 template <typename T>
3 requires somepred<T>() == 42
4 bool foo (T&& lhs, U&& rhs);
5
6 // (2) constexpr function somepred typechecked only
7 template <typename T>
8 requires requires (T t) {
9     somepred<T>() == 42;
10 }
11 bool bar (T&& lhs, U&& rhs);

```

Именно это делает сложные ограничения сложными.

Есть три основных категории сложных ограничивающих выражений:

1. Базовые ограничивающие выражения – проверяют наличие операции над типами и на этом всё.

```

1 template <typename T>
2 requires requires (T a, T b) {
3     a + b;
4 }
5 T test1 (T, T);

```

Есть некое тонкое различие между simple и expression requirements, выше указана первая форма, но это уже очень тонкая разница.

2. Ограничивающие выражения для типов – проверяют наличие типа.

```

1 template <typename T>
2 requires requires() {
3     typename T::inner;
4 }
5 T test2 (T, T);

```

3. Составные ограничивающие выражения – проверяют и операцию и выводимый тип.

```

1 template <typename T>
2 requires requires(T x) {
3     {*x} -> typename T::inner;
4 }
5 T test3 (T, T);

```

Кроме этих ограничений есть ещё ограничения на спецификацию исключений и прочая экзотика, которую можно прочитать в очередном TS, но чтобы составить общее впечатление достаточно различать эти три вида.

Немножко практики. Пусть даны структуры, отвечающие некоторым из упомянутых выше требований.

```

1 struct HasInner {
2     using inner = int;
3 };
4
5 struct HasDeref {
6     using inner = int;
7     inner operator*();
8 };

```

Можно посмотреть какие выполняются и не выполняются в каких случаях.

```

1 int main () {
2     test1(1, 1);
3     test1(HasInner{}, HasInner{}); // no +
4     test2(HasInner{}, HasInner{});
5     test2(1, 1); // no inner
6     test3(HasDeref{}, HasDeref{});

```

```
7     test3(HasInner{}, HasInner{}); // no deref
8 }
```

Разумеется сложные ограничивающие выражения можно комбинировать.

```
1 template <typename T>
2 requires requires(T x) {
3     {*x} -> typename T::inner;
4     } &&
5     requires() {
6         typename T::inner;
7     } &&
8     requires (T a, T b) {
9         a + b;
10    }
11 T test4 (T, T);
```

**Вопрос к студентам:** написать тип, который подойдет под такое ограничение.

Сложные ограничения позволяют очень многое (неожиданно многое) но они имеют несколько сложный синтаксис, поэтому сложные примеры следует отложить до введения простого способа выносить сложные ограничения в общую часть – собственно концептов (мы до них добрались, да).

### 5.11.5 Простые концепты

Выше уже можно было увидеть, что сложные ограничения могут быть очень сложны, а главное – всегда есть желание их переиспользовать, а не писать каждый раз заново.

Например можно прикинуть какой интерфейс должен поддерживать тип, чтобы быть последовательностью с обратной вставкой (Sequence)? На ум приходит следующее.

1. тип `T::element_type` для указания своего типа элементов
2. метод `size()` для проверки размера, возвращающая `size_t`
3. метод `empty()` для проверки пустоты, возвращающий `bool`

4. метод `back()` для получения последнего элемента, возвращающий `T::element_type`
5. метод `push_back()` для вставки в конец последовательности

Простой концепт для такого класса типов можно определить как (это можно пошагово разобрать у доски).

```

1 template <typename T>
2 concept bool Sequence() {
3     return
4         requires { typename T::element_type; } &&
5         requires (T t, typename T::element_type x) {
6             { t.size() } -> size_t;
7             { t.empty() } -> bool;
8             { t.back() } -> typename T::element_type;
9             { t.push_back(x) } // not specified (!)
10        };
11    }

```

Есть несколько форм эквивалентного синтаксиса, в которых можно использовать получившийся концепт:

1. Базовый синтаксис – явное использование в ограничении

```

1 template <typename T>
2 requires Sequence<T>
3 void fill_with_random (T &x, int n);

```

2. Упрощенный синтаксис – вынос констрайнта в шаблонный параметр

```

1 template <Sequence T>
2 void fill_with_random (T &x, int n);

```

3. Синтаксис с выводом типов:

```

1 void fill_with_random (Sequence &x, int n);

```

Последний вариант выглядит несколько странно (и вызывает обоснованную критику), но в языке уже включена в стандарт C++17 возможность эквивалентно писать формы для обычных функций с шаблоном и с `auto`.

1. Форма обычного шаблона функции

```

1 template <typename T>
2 void foo (T x);
3
4 template<typename T, typename U, typename V>
5 void bar(T (U::*)(V));

```

2. Форма с выводом типов

```

1 void foo (auto x);
2
3 void bar(auto (auto::*)(auto));

```

В свете этого идея такого же вывода концептами находится вполне в русле преобразований в язык. Аналогично в языке уже (с 14-го года) существуют обобщенные лямбды. Аналогично нет ничего противоречивого в том, чтобы ограничивать их концептами.

1. Обобщенная лямбда

```

1 find_if(v, [str](const auto& x)
2             { return str == x; });

```

2. Ограниченнная обобщенная лямбда

```

1 find_if(v, [str](const String& x)
2             { return str == x; });

```

Есть некие ограничения на то, как могут быть устроены простые концепты-функции: они всегда возвращают `bool`, никогда не принимают аргументов, состоят только из одного `return` (похоже на старые добрые константно-выраженные функции в C++11) и не могут быть ни объявлены а позже определены, ни сделаны функциями-членами, ни быть друзьями классов. И конечно, в отличии от аргументов `constexpr` функций, аргументы концептов не могут быть ограничены в свою очередь концептами.

Концепты (в рамках COncepts Lite) не проверяют семантику, то есть нельзя проверить, что

```

1 t.empty() == (t.size() == 0)

```

Концепты также не проверяют реализацию: пользователь может заложиться внутри шаблона на неявный интерфейс не предусмотренный концептом

```

1 template<Range R, typename T> bool
2 requires RangeEqComparable<R, T>()
3 in (R const& range, T const& value) {
4     for (auto const& x : range)
5         if (x != value) // OOPS!
6             return false;
7     return true;
8 }
```

Этот пункт вызывает споры и критику. Многие считают, что как раз проверять реализацию шаблона – важное предназначение концептов как инструмента.

В рассмотренном выше примере, концепт `Sequence` состоял только из сложных ограничений, но можно делать концепты и из простых ограничений.

```

1 template <typename C>
2 concept bool isInt() {
3     return std::is_integral<C>::value;
4 }
```

Мало того, концепты вообще не обязаны быть функциями. Самые простые концепты получаются из шаблонных переменных.

```

1 template <typename C>
2 concept bool Int = std::is_integral<C>::value;
```

Это работает так же как обычные шаблонные переменные. Можно записать даже проще.

```

1 template <typename T>
2 concept bool Int = is_integral_v<T>;
3
4 template <typename T>
5 concept bool Float = is_floating_point_v<T>;
```

Здесь использован явный синтаксис присваивания концепту шаблонна переменной (по сути присваивание на метаклассах).

Увы, концепты не поддерживают рекурсию и поэтому не Тьюринг-полны. Также их нельзя специализировать. Зато возможна частичная

специализация по констрайнту:

```

1 template <typename T>
2 class complex;
3
4 template <Float T>
5 class complex { /* complex numbers */ };
6
7 template <Int T>
8 class complex { /* gaussian integers */ };

```

Это похоже на пример с конструкторами, но выглядит ещё более симпатично. Самое интересное: любой (даже тривиальный как C1 выше) констрайнт ограничивая шаблон делает его более специализированным, чем не ограниченный шаблон.

Не менее красиво выглядит теперь и пример с перегрузкой конструкторов:

```

1 struct Foo {
2     enum {int_like, float_like} type_;
3     Foo (Int x);
4     Foo (Float x);
5 };

```

Тем не менее, иногда нетривиальный концепт может быть довольно сложен. Поэтому простые концепты можно писать в терминах ещё более простых. Например для уже рассматривавшейся проверки на сравнимость на равенство можно соорудить вот такое:

```

1 template <typename T, typename U>
2 concept bool Weak_equality_comparable() {
3     return requires(T t, U u) {
4         {t == u} -> bool;
5     };
6 }
7
8 template <typename T, typename U = T>
9 concept bool Equality_comparable() {
10    return Weak_equality_comparable<T, U>() &&
11        Weak_equality_comparable<U, T>();
12 }

```

Но вот в том как воспользоваться таким концептом для определения функции `check()` есть тонкости.

**Вопрос к студентам:** сработает ли вот такой вариант?

```
1 template <Equality_comparable T,
2           Equality_comparable U>
3 bool check(T, U);
```

Для решения этой задачи можно использовать либо явный синтаксис с ограничением, либо принципиально новый синтаксический выверт – введение шаблона.

```
1 Equality_comparable{T, U}
2 bool check(T, U);
```

Эта запись является сокращением для

```
1 template <typename T, typename U>
2 requires Equality_comparable<T, U>()
3 bool check(T, U);
```

И выглядит довольно интересно. Также столь удобный синтаксис позволяет заменить `auto` на концепт:

```
1 template <typename C>
2 concept bool Int = std::is_integral<C>::value;
3
4 // ...
5
6 Int x = f(x); // auto x = f(x);
```

Благодаря концепту теперь в коде будет яснее чего ожидать от вывода типов (ну и усилится проверка от человеческих ошибок).

### 5.11.6 Вариабельные концепты

Ещё один пример полезного использования получившегося ограничения `Sequence` это подсчет количества элементов, удовлетворяющих предикату. Первый вариант может выглядеть как-то так.

```
1 template <Sequence S, typename P>
2 int count (S const& seq, P pred) {
3     int n = 0;
```

```

4   for (auto const &x : seq)
5     if (pred(x)) ++n;
6   return n;
7 }
```

Увы, он очевидно underconstrained (русское слово недоограничен звучит не менее ужасно). Вместо второго параметра может быть передано все что угодно. Чтобы это исправить, нужно ввести явное ограничение, например так:

```

1 template <Sequence S, typename P>
2   requires Predicate<P, typename S::element_type>()
3 int count (S const& seq, P pred) {
4   int n = 0;
5   for (auto const &x : seq)
6     if (pred(x)) ++n;
7   return n;
8 }
```

Но что такое `Predicate`? Очевидно это функция, которая может быть вызвана с аргументом, заданного типа. Идея заключается в том, что аргументов может быть сколько угодно, поэтому можно воспользоваться вариабельными шаблонами:

```

1 template <typename P, typename ... Args>
2 concept bool Predicate() {
3   return requires (P pred, Args ... args) {
4     { pred(args...) } -> bool;
5   };
6 }
```

Эта идея показывает гибкость концептов, в которых возможны даже вариабельные шаблоны. Более того: синтаксис введения шаблонов также вполне уживается с троеточиями.

```

1 template<typename T, int N, typename... Xs>
2 concept bool C1 = true;
3
4 C1{A, B, ...C} struct S1
```

Такой синтаксис введения шаблона по понятным причинам подвержен ещё большей критике.

### 5.11.7 Частичное упорядочение

Выше показаны случаи, когда компилятору необходимо разрешить перегрузку по концептам. Для этого концепты должны быть сравнимы (чтобы выбрать более подходящие и менее подходящие).

Говорят, что концепт P поглощает концепт Q если можно доказать, что из P следует Q

Каждый концепт в рамках реализации Concepts Lite разбивается на элементарные ограничения, соединенные конъюнкциями или дизъюнкциями. После этого они нормализуются по следующим правилам: Нормализация предикатного констрайнта – его полная подстановка подвыражений. Вместо каждого вызова концепта-функции подставляется тело, вместо каждого обращения к концепту переменной подставляется определяющее его выражение. Все найденные в процессе этого длинные логические операции формируют конъюнкции и дизъюнкции. Нормализация любой конъюнкции и дизъюнкции это нормализация её operandов. Requires с непустым телом трансформируется в параметризованный концепт.

После такого разбиения, концепт P переводится в дизъюнктивную, а Q в конъюнктивную нормальную форму. Если в конъюнкте  $P_i$  каждая переменная поглощает все переменные дизъюнкта  $Q_j$ , то  $P_i$  поглощает  $Q_j$ . Если каждый коньюнкт из ДНФ для P поглощает любой дизьюнкт из КНФ для Q, то P поглощает Q.

Пример двух концептов, один из которых поглощает другой.

```

1 template<typename T>
2 concept bool Subsumed() {
3     return requires () { typename T::type1; };
4 }
5
6 template<typename T>
7 concept bool Subsuming() {
8     return Subsumed<T>()
9     && requires () { typename T::type2; };
10 }
```

**Вопрос к студентам:** как будет выглядеть Subsuming после нормализации?

Сплиттинг типа по этим концептам.

```

1 template<typename T>
2 struct TM;
3
4 template<Subsumed T>
5 struct TM<T> { TM() {std::cout << "Subsumed!\n";} };
6
7 template<Subsuming T>
8 struct TM<T> { TM() {std::cout << "Subsuming!\n";} };

```

Применение (весьма условно полезное)

```

1 struct M {
2     using type1 = int;
3     using type2 = int;
4 };
5
6 TM<M> X{};

```

**Вопрос к студентам:** что будет на экране?

### 5.11.8 Критика концептов

Основные направления критики концептов со стороны комитета по стандартизации делятся на идеологические и технические.

Идеологические.

- Предложение опубликовано менее года назад и спецификация реализована только в одном компиляторе и реализована тем же человеком, который писал документ спецификации. В итоге может получиться как с экспортом шаблонов, который тоже был реализован в одном компиляторе. Никто этого не хочет.
- Как следствие – сейчас очень мало кода, который полагался бы на концепты или хоть как-то их использовал. Таким образом может получиться что будут забыты или не учтены важные use-cases, а потом стандарт отольёт все эти ошибки в границе.

Технические.

- Синтаксис `void f(X x){}` определяет функцию если X тип, но шаблон функции если X концепт. Это может привести к пугающим синтаксическим неоднозначностям.

- Синтаксис введения шаблонного параметра `C{A,B} void f(A a, B b)` вызывает массу вопросов (например перегружает дополнительным значением фигурные скобки). Многие в комитете требуют убрать эту возможность.
- Использование концепта требует информации был ли он определен как функция или как переменная. Это вызывает неоднозначности в использовании и по мнению комитета создает преграду для обучения.
- Для C++17 уже утвержден синтаксис для шаблонных параметров не являющихся типами:

```

1 template<auto V>
2 constexpr auto v = V*2;
```

Но если кто-то попробует ограничить `V` констрайнтом, такое ограничение будет конфликтовать с синтаксисом для шаблонного ограничения на тип (например `Integral`).

- Ошибки проверки концептов часто проявляются как ошибки разрешения перегрузки, что, по мнению комитета, может не упрощать, а усложнять сообщения компилятора об ошибках (много сообщений об отвергнутых кандидатах с объяснением причин).

И наконец самое странное направление критики – многим членам комитета кажется, что текущие `concepts lite` это шаг не в том направлении, де-факто сливающий те концепты, о которых все на самом деле мечтали.

Чтобы понять это направление, нужно несколько откатиться в историю.

### 5.11.9 Сны о чем-то большем

Концепты в текущей ревизии не зря называются `concepts lite`. Изначально концепты рассматривались как гораздо более тяжелая и мощная надстройка над языком.

Вот пример из статьи Страуструпа и Саттона 2011 года, показывающий концепты мечты

```

1 concept Comparable<typename T> {
2     // syntax of equality
```

```

3   requires constraint Equal<T>;
4
5   // semantics of equivalence
6   requires axiom Equivalence_relation<Equal<T>, T>;
7
8   // if x == y then for any Predicate p, p(x) == p(y)
9   template<Predicate P>
10  axiom Equality(T x, T y, P p) {
11      x == y => p(x) == p(y);
12  }
13
14  // inequality is the negation of equality
15  axiom Inequality(T x, T y) {
16      (x!=y) == !(x==y);
17  }
18 }
```

Концепт comparable определяет понятие равенства. Он требует оператор `==` через констрайнт `Equal`, а семантическое значение этого предиката задается аксиомой `Equivalence_relation` (определяет рефлексивность, симметричность и транзитивность). Дополнительные две аксиомы определяют семантический смысл равенства и обратного ему неравенства.

Кроме того, такой концепт ещё должен был проверять реализацию шаблона, который был им ограничен.

Конечно, это никто никогда не мог реализовать.

Также в раздел не сбывшихся пока мечт (но уже близких к), можно добавить контракты. К сожалению, TS по контрактам ещё не реализовано в GCC и поэтому они будут рассмотрены только вкратце.

Контракты это по сути прокачанные asserts.

```

1 void
2 MyVector::push_back(Elem e) [[ensures: data != nullptr]]
3 {
4     if(size >= capacity)
5     {
6         Data* p = data;
7         data = nullptr; // Just for the sake of the example...
8         data = new Data[capacity*2]; // Might throw an
9         exception
10    }
```

```
9      // Copy p into data and delete p
10     }
11     // Add the element to the end
12 }
```

Если конструктор выбросит исключение, постусловие контракта не будет соблюдено.

Кроме ключевого слова `ensures`, означающего проверку на выходе, контракты также поддерживают ключевое слово `expects` для проверки на входе.

```
1 T& MyVector::operator[](size_t i) [[expects: i < size()]];
```

Расширяя обычные асsertы, контракты разумеется могут быть отключены (причем в TS по контрактам предусмотрено как отключение только предусловий, так и отключение только постусловий).

## 5.12 Домашняя наработка по стандартной библиотеке

### Контрольные вопросы

1. Какой тип имеет строковый литерал "Hello, world"?
2. С-строка `cheap` выделена на куче и имеет размер 5. Какие опасности создаст присвоение `cheap = "Hello, world"`?
3. В чём преимущества и в чём недостатки `std::string` по сравнению с С-строкой
4. В чём преимущества и в чём недостатки `std::string_view` по сравнению с С-строкой
5. Вам нужна статическая неизменная строка в вашей программе. Например “sator arepo tenet opera rotas”. Она должна быть глобальной. Какой тип вы для неё выберете и почему?
6. Почему `remove_prefix` и `remove_suffix` существуют для `string_view`, но не для `string`?
7. Что такое COW string, какие у этих строк преимущества и какие проблемы?
8. Что такое SSO string, какие у этих строк преимущества и какие проблемы?
9. Каким образом в строках учитываются разные типы / разная ширина символов?
10. Старый добрый С даёт массу возможностей обработки ошибок с помощью кодов возврата. Зачем вообще в C++ взялись выдумывать нечто новое?
11. Что является и что не является исключительной ситуацией в терминах C++?
12. Чем неконсистентное состояние объекта отличается от неопределенного? Приведите пример.
13. Почему не рекомендуется использовать в качестве объектов исключений объекты встроенных типов, вроде `int` и `float`?

14. Что в практическом смысле означают слова "бросить исключение" и "поймать исключение"?
15. Почему не рекомендуется ловить исключения по значению?
16. Что такое "размотка стека" и как она происходит?
17. Может ли при размотке стека после исключения возникнуть висячая (dangling) ссылка? Приведите пример. Какие проблемы могут возникнуть если в вашей иерархии классов для объектов исключений вы использовали множественное наследование?
18. У вас есть выбор – ловить исключение по ссылке или по указателю. Какие аргументы за и против вы приведете и что выберете?
19. Что произойдет если при создании объекта исключения, его конструктор выбросил исключение?
20. Можно ли бросить и поймать объект исключения с запрещенным копированием и перемещением? А перевыбросить?
21. Что такое нейтральность относительно исключений и каким образом перехват всех исключений нарушает гарантию нейтральности?
22. Какие преимущества и недостатки вы видите при использовании исключений вместо кодов возврата?
23. Что такое небезопасный относительно исключений код? Приведите примеры.
24. Какие гарантии безопасности исключений вы знаете и что такое базовая гарантия?
25. Каким образом безопасность исключений влияет на проектирование кода?
26. Какие формы оператора new вы знаете и что такое размещающий new?
27. Почему исключения не должны покидать деструктор и что будет если всё же вдруг.
28. Может ли предоставлять базовую гарантию исключений класс, управляющий двумя ресурсами, как именно или почему нет?
29. Что такое аннотация `noexcept` и оператор `noexcept`?

30. Расскажите про оборачивание исключений в `exception_ptr` и про вложенные исключения.
31. Можно ли в деструкторе определить был он вызван регулярным образом или в процессе размотки стека при исключении?
32. В списке инициализации конструктора два указателя инициализируются через `new`. Как обеспечить чтобы оба были либо полностью выделены либо освобождены и конструирование всего объекта выбросило исключение?
33. По ссылкам на каких предков (левого, правого или общего) можно поймать в обработчике исключения объект самого нижнего производного класса при ромбовидном наследовании?
34. Можно ли для класса с динамическим выделением одного из полей как-то гарантировать, что исключения не покинут копирующий конструктор?
35. Copy and swap idiom – что это такое, применимость, примеры
36. Почему в стандартной библиотеке метод `vector<T>::pop_back()` не возвращает значение удаленного из вектора элемента? (Подсказка: это как-то связано с безопасностью относительно исключений).
37. Что такое `auto_ptr` и чем опасно его размещение в контейнерах?
38. Почему `unique_ptr` хитро специализирован для массивов?
39. Возможен ли `unique_ptr<void>`?
40. Что такое пользовательский удалитель и почему указатель на функцию хуже, чем класс с переопределённым `operator()`?
41. Что вы думаете о передаче лямбда-функции как пользовательского удалителя в `unique_ptr`? Почему это лучше, чем передать туда указатель на функцию и как это работает?
42. В чём преимущества и недостатки `shared_ptr` по сравнению с `unique_ptr`?
43. Согласны ли вы с идеей, что `shared_ptr` ничем не лучше, чем глобальная переменная?
44. Зачем можем быть нужны `unique_ptr<const T>` и `shared_ptr<const T>`?

45. В каких случаях прямой конструктор `shared_ptr` работает, а `make_shared` – нет?
46. Что такое `weak_ptr` и почему его нельзя разыменовать?
47. Что такое aliasing constructor для `shared_ptr`? Приведите примеры использования.
48. Как в программе может быть создано два контрольных блока для двух разных `shared_ptr`, управляющих одним ресурсом и чем это чревато?
49. Можно ли из нестатического метода класса вернуть `shared_ptr` на объект, для которого вызван метод?
50. Что такое `static_pointer_cast` и зачем он нужен?
51. Почему на самом деле внутри контрольного блока `shared_ptr` размещается не один, а два счётчика ссылок?
52. Приведите аргументы за и против использования `make_shared`?
53. Как вы относитесь к идеи заменить `std::string` на `unique_ptr<char[]>`?
54. Как формулируются требования к объектам, которые можно положить в контейнер стандартной библиотеки?
55. Вектор предоставляет гарантию непрерывности по памяти. Означает ли это, что указатель на середину вектора можно привести к вектору?
56. Могут ли два контейнера стандартной библиотеки использовать общую область памяти?
57. Расскажите про `vector<bool>`. В чём его особенности, чем он хорош или плох.
58. Чем отличается `size` от `capacity` для векторов? Расскажите про `resize`, `reserve` и про управление памятью вектора.
59. Как работает инициализация фигурными скобками в C++11 и далее? Что такое списочная инициализация, что такое расширенный синтаксис инициализации, в чём разница.
60. В чём разница `std::array` со встроенными массивами?

61. В каких случаях вы предпочтёте `deque`, а в каких `vector`?
62. Что такое сплайс и как он работает для односвязных и двусвязных списков?
63. Расскажите про контейнерные адаптеры, какие из стандарта вы сможете вспомнить
64. Расскажите про `valarrays`, особенно интересно что такое слайс.
65. Допустим вам нужно хранить битовую маску в программе. Расскажите о выборе между просто `unsigned` типом, `vector<bool>` и `std::bitset`. В каких случаях что вы предпочтёте?
66. Расскажите про `array_view` и чем именно он лучше и так легковесного `attra`. Для чего вы будете использовать `array_view` в вашем коде?
67. Покритикуйте C-style ввод и вывод. Чем плохи старые добрые `printf` и `scanf`?
68. Чем отличается поток вывода от устройства вывода (например от файла или консоли)?
69. Зачем нужна функция `ignore` у потоков ввода и какие проблемы она позволяет решать?
70. Потоки, как ввода так и вывода, дают выбор между форматированным и неформатированным доступом. В каком случае какой вы выберете?
71. Как можно настроить потоки ввода-вывода таким образом, чтобы о приходе в невалидное состояние сообщало исключение и что это будет за исключение?
72. Имеют ли смысл копирование и перемещение для объектов потоков ввода-вывода?
73. Какие особенности и подводные камни имеют стандартные потоки ввода-вывода при работе с бинарными файлами?
74. При форматированном вводе в `std::string` конец строки определяется по разделителю (пробел, перенос строки). Предложите способ добавить произвольный символ в список разделителей.

75. Что такое “локаль” и “фасет”, зачем они нужны?
76. Расскажите про механизм `iword/xalloc` в потоках, зачем он нужен.
77. Что такое сцеплённые (`tied`) потоки?
78. Что такое диапазоны и почему в стандартной библиотеке все диапазоны полуоткрытые?
79. Какие вы знаете категории итераторов и как определить категорию в обобщённом коде?
80. Что такое сингулярный итератор и когда он не является ошибкой?
81. Итераторы различаются в частности по константности и направлению (что даёт четыре степени свободы). Опишите процесс приведения неконстантного обратного к константному прямому итератору и обратный процесс.
82. Как перегружены инкремент и разыменование у `insert`-итераторов и почему выбрано это решение.
83. Обычные итераторы ничего не знают о своём контейнере. Могут ли `insert`-итераторы ничего не знать о своём контейнере? Если да то как этого добиться, если нет, то почему нет.
84. В каких случаях итератор невалиден. Что такое инвалидация итераторов?
85. Расскажите про идиому `erase-remove` и зачем она нужна.
86. В каких случаях `transform` предпочтительней, чем `for_each`?
87. Что такое `move`-итераторы и в каких случаях их используют?
88. Каким условиям должны отвечать предикаты для ассоциативных контейнеров. Приведите примеры нарушения каждого.
89. Как правильно удалить каждый второй элемент из множества `std::set`? В чём тут могут быть сложности?
90. Расскажите о разных способах добавить пару ключ-значение в отображение, сравните их преимущества и недостатки.
91. Объясните как использовать мультиотображение (или несколько мультиотображений) для моделирования направленного ациклического графа.

92. Расскажите как вы будете выбирать между упорядоченными и неупорядоченными множествами в вашем коде. В каком случае вы предпочтёте обойтись просто сортированным вектором не связываясь с множествами вообще?

**Задания:**

1. Напишите для `std::string` простой аналог сишной функции `strtok`
2. Напишите на C++ функцию `combine`, которая берёт много маленьких строк (произвольное кол-во аргументов) и эффективно их комбинирует. Почему использование в ней оператора `+` будет плохой идеей?
3. Спроектируйте класс строк, работающих лениво, в том смысле, что при копировании такой строки копия разделяет тот же буфер и только при изменении одной из копий реально создаётся копия буфера.
4. Спроектируйте собственный простой `scoped pointer`. В чём его проблемы?
5. Спроектируйте собственный простой `unique pointer`. В чём его проблемы?
6. Спроектируйте дерево, состоящее из уникальных указателей. Как будет выглядеть удаление поддерева в этом дереве?
7. Напишите простейший `intrusive_ptr` с хранением счётчика ссылок на уровне объекта и обязательным интерфейсом `addref` и `release` у хранимых сущностей.
8. Спроектируйте (без стандартной библиотеки) шаблонный класс очереди, используя односвязный циклический список.
9. Спроектируйте простейшую иерархию классов с виртуальным копированием (например с виртуальным `clone`, позволяющим создать копию потомка по указателю на базовый класс). Можно ли как-то упростить написание наследников, используя CRTP?
10. Спроектируйте класс, представляющий собой пирамиду в непрерывной памяти. Можно использовать стандартную библиотеку.

11. Спроектируйте собственный класс односвязного списка `slist`, для которого будут возможны вставка в начало, в конец и `size` за  $O(1)$ , при этом сложность сплайса может быть  $O(N)$ . В чём будет его основное отличие от `std::forward_list`?
12. Спроектируйте собственный адаптер `cyclicbuf` над последовательными контейнерами, представляющий собой циклический буфер (т.е. когда он достигает конца, заполнение продолжается сначала) с размером известным на этапе компиляции. Какой минимальный интерфейс вы для него оставите?
13. У вас есть три функции `top()` а также `left(node_handler)` и `right(node_handler)` возвращающие класс `node_handler`, про который известно только то, что у него есть метод `data`. Эти три функции задают неявное бинарное дерево. Напишите обход такого дерева в ширину и в глубину с помощью `std::stack` и `std::queue`
14. Вам задан файл в формате "`NAME( NUMBERS)*`" где NAME это не содержащая пробела строка любой длины, а NUMBERS это числа разделённые пробелами. При этом чисел может вообще не быть. Далее следует перевод строки. Напишите функцию считывания этого файла в `multimap (string => int)`.
15. Вам задан файл в формате "`(NAMES )*NUMBER`" где NUMBER это число, а NAMES это не содержащие пробелов строки любой длины, разделённые пробелами. При этом строк может вообще не быть. Далее следует перевод строки. Напишите функцию считывания этого файла в `multimap (int => string)`.
16. Разработайте собственный модификатор `mutate` для потоков вывода, который заменяет все английские а на русские а. Он должен заливать, то есть:

```
1 cout << mutate << "Alloha "; cout << "anyone\n";
```

должно заменить а и при втором выводе. Разработайте в пару к нему `unmutate` чтобы отменять его действие.
17. Напишите адаптер `randomize_cont` чтобы с ним range-based обход обходил все элементы контейнера в случайному порядке.

```
1 for (auto x : randomize_cont(v))
```

18. Напишите алгоритм, который собирает элементы вокруг определённой позиции

`1 gather(start, finish, pos, predicate).`

Например если предикат `x % 2 == 0` то массив `1 2 3 4 5 6 7 8` собранный возле позиции элемента 4 должен стать `1 3 2 4 6 8 5 7`. Постарайтесь сделать его в терминах других алгоритмов стандартной библиотеки.

19. Написать функцию которая берёт `multimap`, сопоставляющий класс (целое число) значению (строка), например `1 => lion, 2 => pear, 1 => bear, 3 => ford, 2 => carrot` и переводит его в `vector<vector<string>>` где номер класса это индекс во внешнем векторе. В данном случае это будет  `{{lion, bear}, {pear, carrot}, {ford}}`

# Глава 6

## Толкаясь невидимыми локтями

*I've written well over a thousand programs,  
many of which have substantial size.  
I can't think of even five of those programs  
that would have been enhanced noticeably  
by parallelism or multithreading*

– Donald Knuth

Многопоточность долгое время оставалась за кадром стандартных возможностей C++ и была отдана на откуп API конкретных библиотек и операционной системы. С одной стороны это породило тонны чудовищного легаси, когда в одной программе под условной компиляцией сочетались windows и posix потоки. С другой стороны это позволило отложить проектирование многопоточности в языке и сразу сделать всё более-менее правильно. Начиная с 2011 года многопоточность уверенно вошла в язык, принеся с собой много новых понятий и концепций.

В это части сделана попытка изложить не просто языковые средства многопоточности, но дать представление о самой идее, о специфичной логике таких программ и об основных решениях и идиомах при их проектировании.

## 6.1 Нити исполнения и вышивание

Существует своеобразная терминологическая омонимия в русском слове “поток”. Им обозначают как поток ввода-вывода (то что по английски называется stream) так и нить исполнения программы (то что по английский называется thread). Эта путаница не так страшна, поскольку в большинстве случаев всё вполне понятно из контекста. Тем не менее, иногда в этой главе будут использоваться слово “нить” или англизм “тред” позволяющие избежать двусмыленности.

Первая программа hello world как раз задействует потоки в обоих смыслах: создаётся нить исполнения, которая выводит приветствие миру на стандартный поток вывода.

```
1 int main() {
2     thread t([]{ cout << "Hello, world!" << endl; });
3     t.join();
4 }
```

Две вещи которые вызвавший контекст может сделать с порождённым потоком это дождаться его (`t.join()`) или отказаться от него (`t.detach()`). Если не сделать ни того ни другого, то по завершению функции (скажем `main` в примере выше), будет вызвана функция `std::terminate` и исполнение программы будет аварийно завершено.

**Вопрос к студентам:** что это означает с точки зрения безопасности относительно исключений?

К сожалению, нет никакого простого стандартного способа прервать поток извне. В стандарте POSIX для этого используются сигналы, но никаких сигналов в мире C++ не существует. Да, всегда можно организовать снутри потока какой-то схожий механизм руками, но это уже не то.

Важно понимать, что функция `main` тоже выполняется в потоке (это основная нить выполнения программы и она неявно присутствует всегда).

```
1 int main () {
2     cout << "Parent id: " << std::this_thread::get_id() << endl;
3     thread t([] {
4         cout << "Spawned id: "
5             << std::this_thread::get_id() << endl;
6     });
}
```

```

7     t.join();
8 }
```

Нити исполнения естественным образом возникают в различных задачах: например пользовательский интерфейс может работать в одном потоке, пока в другом потоке предвычисляется содержимое его выпадающих списков на случай если пользователь решит их раскрыть. Или некий алгоритм (такой как сортировка) является ключевым в программе и для него хотелось бы использовать все доступные аппаратные возможности: тогда сортировку можно прогнать во много потоков а потом смержить результаты.

В целом параллельное программирование нужно для производительности, особенно в современных обстоятельствах, когда большинство процессоров стали многоядерными (и даже поддерживающими много аппаратных потоков).

В принципе, многопоточность (concurrency) не требует параллелизма. Пример с потоком для GUI естественно возникает в системах с одним аппаратным потоком и первые многозадачные ОС появились в них же. До конца этой части, впрочем, будет рассматриваться именно параллелизм и его организация для производительности, а прочие аспекты, такие как синхронизация обращений будут рассмотрены начиная с (6.2).

### 6.1.1 Параллельное выполнение

Разумеется, чтобы распараллелить какую-нибудь простую задачу, например сортировку или суммирование массива, в потоки надо научится передавать данные и возвращать их.

Здесь всё довольно просто: функция потока принимает произвольное количество аргументов, которые передаются потоку при его создании. Если аргументы нужно передать по ссылке, они заворачиваются в обёртку вроде `std::ref`.

```

1 void divide(int& result, int a, int b) {
2     result = a / b;
3 }
4
5 int main () {
6     int result;
7     thread t(divide, std::ref(result), 30, 6);
8     t.join();
```

```
9     cout << "result: " << result << endl;
10 }
```

И вот пришло время посмотреть как многопоточность дарит нам производительность.

Распараллеливание суммирования элементов массива рассмотрено в большом количестве источников, в частности [16].

Приведённый ниже код неоправданно прост и наивен и всё равно по замерам автора даже на слабеньком core i3 он даёт существенный прирост: 0.15 против 0.2 секунд для массива в 200 миллионов целых.

```
1 template <typename Iterator, typename T>
2 T parallel_accumulate(Iterator first, Iterator last, T init)
3 {
4     long length = distance(first, last);
5     if (0 == length) return init;
6     unsigned nthreads = 4;
7     long bsize = length / nthreads;
8
9     vector<thread> threads(nthreads);
10    vector<T> results(nthreads + 1);
11    auto accumulate_block = [](Iterator first,
12                                Iterator last, T& result) {
13        result = accumulate(first, last, T{});
14    };
15
16    unsigned tidx = 0;
17    for (; length >= bsize * (tidx + 1);
18          first += bsize, tidx += 1)
19        threads[tidx] = thread(accumulate_block,
20                               first, first + bsize,
21                               ref(results[tidx]));
22
23    auto remainder = length - bsize * tidx;
24    if (remainder > 0) {
25        assert(tidx == nthreads);
26        results[tidx] =
27            accumulate_block(first, first + remainder, T{});
28    }
29
30    for (auto&& t: threads)
```

```

31     t.join();
32     return accumulate(results.begin(), results.end(), init);
33 }
```

Хвост массива обрабатывается в том же потоке, что даёт время порождённым потокам завершить работу и не оставляет основной поток ждать без дела.

На строчке 6 здесь определяется количество потоков и это делается довольно глупым образом: берётся просто число 4. В принципе это довольно пессимистично для сильных компьютеров, имеющих много потоков и, вместе с тем, сильно оптимистично для какой-нибудь embedded железки.

```

1 unsigned determine_threads(unsigned length) {
2     const unsigned long min_per_thread = 25;
3     unsigned long max_threads = length / min_per_thread;
4     unsigned long hardware_conc = std::thread::
5         hardware_concurrency();
6     return std::min(hardware_conc != 0 ? hardware_conc : 2,
7                     max_threads);
}
```

В этой функции тоже задаётся некий произвольный параметр: минимальное число элементов в подмассиве чтобы ради них вообще имело смысл вызывать отдельный поток, но этот параметр имеет куда больше смысла. Остальное тут определяется через функцию `hardware_concurrency`, позволяющую оценить возможности текущей аппаратуры к параллелизму.

Далее использование становится довольно очевидным.

```
1 unsigned nthreads = determine_threads(length);
```

**Домашняя наработка:** нагуглите и реализуйте распараллеливание быстрой сортировки.

Для C++20 сейчас рассматривается Parallelism TS – параллельные версии алгоритмов для включения в стандартную библиотеку. Во многих современных компиляторах они уже включены в пространство имён `experimental` и доступны для того, чтобы с ними играть.

Например параллельный `accumulate` это `parallel::reduce` и его во все не нужно писать руками.

```
1 std::experimental::parallel::reduce(
```

```

2   std::experimental::parallel::par,
3   v.begin(), v.end());

```

**Домашняя наработка:** исследуйте, поддерживает ли ваш компилятор параллельный reduce? Что говорят замеры против реализованного вручную?

Кроме того специальное полустандартное расширение OpenMP позволяет параллелизовать код в том числе нетривиальный, посредством установки специальных прагм в нужных местах, а остальное компилятор делает за пользователя.

Интенсивный курс по OpenMP можно найти здесь: [https://courses.cs.washington.edu/courses/csep524/13wi/omp\\_tutorial\\_full.pdf](https://courses.cs.washington.edu/courses/csep524/13wi/omp_tutorial_full.pdf)

### 6.1.2 Обещания и асинхронность

Распараллеливание accumulate, sort и прочих алгоритмов может быть рассмотрено и с другой точки зрения: среда исполнения получает ряд задач, которые асинхронно исполняются. Часть из них могут в принципе исполняться и на одном и том же потоке, программисту не всегда следует лезть в этот механизм руками. Стиль языка C++ всегда гарантирует, что если что, то залезть руками можно, но всегда надо понимать, что более высокий уровень абстракции означает меньшее головной боли.

Проблема, как обычно в том, чтобы обобщить коммуникацию между потоками на коммуникацию между задачами. И здесь в игру вступает механизм обещаний.

Функция деления из (6.1.1) выглядела не слишком красиво

```

1 void divide(int& result, int a, int b) {
2     result = a / b;
3 }

```

Канал коммуникации есть, но он какой-то уж очень самобытный. Вместо этого можно использовать передачу в функцию обещания

```

1 void divide(promise<int> result, int a, int b) {
2     result.set_value(a / b);
3 }

```

Теперь специальная сигнализация через `std::future` позволяет использовать результат ровно тогда, когда он готов

```

1 promise<int> result;
2 future<int> f = result.get_future();
3 thread t(divide, move(result), 30, 6);
4 t.detach();
5
6 // .... more code ...
7
8 cout << "result: " << f.get() << endl;

```

Если к моменту вызова `f.get()` результат всё ещё не будет готов, вызвавший эту функцию поток заблокируется до тех пор, пока не появится результат (в частности пока поток не выставит значение объекту обещания). Это, разумеется, создаёт непаханное поле для всякого рода мёртвых блокировок.

**Вопрос к студентам:** перечислите нетривиальные способы навечно заблокировать поток с помощью механизма `std::future`.

Объект `std::promise` позволяет вызывать два метода для установки результата: `set_value` и `set_exception`. В случае если установлено исключение, оно выстрелит при попытке сделать `future::get`. Это очень похоже на механизм `expect`, предлагаемый в новый стандарт для исключений.

Чтобы не работать с обещаниями руками, можно завернуть функцию, возвращающую результат, в специальную обёртку, `packaged_task` заточенную под использование такого рода

```

1 int divide(int a, int b) {
2     return a / b;
3 }
4
5 packaged_task<int(int, int)> task {divide};
6 future<int> f = task.get_future();
7 thread t(move(task), 30, 6);
8 t.detach();
9
10 // .... more code ...
11
12 cout << "result: " << f.get() << endl;

```

Теперь упакованная задача позволяет не портить саму функцию `divide`, а просто завернуть её в класс и отдать потоку. На самом деле можно рассматривать `std::packaged_task` как своего рода активную версию

`std::function`. Он также инкапсулирует функцию и способен принимать аргументы, только вместо результата у него – обещание результата.

В качестве примера, можно переписать параллельный accumulate из (6.1.1) с помощью `packaged_task`

```

1 template <typename Iterator, typename T>
2 T parallel_accumulate(Iterator first, Iterator last, T init)
3 {
4     long length = distance(first, last);
5     if (0 == length) return init;
6     const unsigned nthreads = determine_threads(length);
7     long bsize = length / nthreads;
8
9     vector<future<T>> results(nthreads);
10    auto accumulate_block = [](Iterator first, Iterator last) {
11        return accumulate(first, last, T{});
12    };
13
14    unsigned tidx = 0;
15    for (; length >= bsize * (tidx + 1);
16          first += bsize, tidx += 1) {
17        packaged_task<T(Iterator, Iterator)> task{accumulate_block
18            };
19        results[tidx] = task.get_future();
20        thread t {move(task), first, first + bsize};
21        t.detach();
22    }
23
24    auto remainder = length - bsize * tidx;
25    T result = init;
26    if (remainder > 0) {
27        assert(tidx < nthreads);
28        result += accumulate_block(first, first + remainder);
29    }
30
31    for (unsigned long i = 0; i < nthreads; ++i)
32        result += results[i].get();
33    return result;
34 }
```

Основные особенности: теперь не надо хранить массив потоков, о

каждом потоке можно просто забыть через `detach` и это будет окей, потому что от него уже получено обещание и сохранено в массив обещаний. Теперь финальный сбор `results[i].get()` подождёт сколько надо до готовности результата упакованной в поток задачи.

Ещё более удобно вообще не использовать порождение потока в явном виде, а сразу отдать задачу на асинхронное выполнение.

Синтаксический сахар для этого предоставляет `std::async`

В итоге сравнительно сложный для понимания код, явно порождающий потоки и отдающий в них задачи в ручном режиме

```

1 for (; length >= bsize * (tidx + 1);
2     first += bsize, tidx += 1) {
3     packaged_task<T(Iterator, Iterator)> task{accumulate_block};
4     results[tidx] = task.get_future();
5     thread t {move(task), first, first + bsize};
6     t.detach();
7 }
```

Превращается в совсем простой:

```

1 for (; length >= bsize * (tidx + 1);
2     first += bsize, tidx += 1)
3     results[tidx] = async(accumulate_block,
4                           first, first + bsize);
```

Замеры скорости не показывают существенных проседаний, все эти абстракции вполне прозрачны и бесплатны.

Для сравнения, решение в стиле OpenMP будет выглядеть очень похоже (в коде ниже `results` это уже не массив `futures`, а просто массив целых чисел).

```

1 for (; length >= bsize * (tidx + 1);
2     first += bsize, tidx += 1) {
3     #pragma omp task
4     results[tidx] = accumulate_block(first, first + bsize);
5 }
6 . . . etc . .
7 #pragma omp taskwait
```

Ещё одно полустандартное расширение языка, реализованное в компиляторе ICC компании Intel и частично поддержанное в GCC называется Cilk. Можно назвать Cilk конкурирующим с OpenMP стандартом и,

к сожалению, Cilk исторически проигрывает этот бой, хотя по мнению автора этих лекций, он даже несколько более совершенен.

Для сравнения тот же самый код на Cilk будет выглядеть следующим образом:

```

1 for (; length >= bsize * (tidx + 1);
2     first += bsize, tidx += 1) {
3     results[tidx] = cilk_spawn accumulate_block(first, first +
4         bsize);
5 }
5 .... etc ...
6 cilk_sync

```

Сама идея пользоваться полустандартными расширениями когда у нас есть стандартный способ сделать тоже самое выглядит странной, но вообще-то обычно именно для параллелизма, расширения вроде omp и cilk или даже библиотеки такие как ppl или tbb работают на пуле потоков, не создавая их каждый раз (что дорого) и отлично оптимизированы под аппаратуру. В некоторых докладах, например [43] сообщается, что производительность, если речь идёт о чистом параллелизме, может отличаться от самодельных решений **в разы**.

И это не только о производительности. С использованием Cilk возможен вот такой финт

```

1 template <typename Iterator, typename T>
2 T parallel_accumulate(Iterator first, Iterator last, T init)
3 {
4     T result {};
5     if (last - first < 32) {
6         for (auto i = first; i != last; ++i)
7             result += *i;
8     }
9     else {
10         Iterator mid = first + (last - first) / 2;
11         T a = cilk_spawn parallel_accumulate(first, mid, init);
12         T b = parallel_accumulate(mid, last, init);
13         cilk_sync;
14         result = a + b;
15     }
16     return result;
17 }

```

На строчке `cilk_spawn` видно, что её нельзя так просто переписать с помощью `async`, потому что `cilk_spawn` не заботится о потоках, работая на своём пуле, а `async` в большинстве реализаций будет слепо создавать поток за потоком.

Однако, несмотря на сомнительность некоторых примеров, сам механизм `future / promise` объективно полезен и имеет много приложений.

### 6.1.3 Чуть больше про обещания

Обещание, которое никто не собирается выполнять – простой способ навсегда заблокировать поток.

```
1 promise<int> pr;
2 auto fut = pr.get_future();
3 fut.get(); // forever
```

С этим связана опасность исполнения `async` на пулах потоков: если два зависимых обещания пула распределит в один поток, эта программа не завершится. К сожалению, в стандарте не специфицировано породит ли каждый вызов `async` новый тред или будет использован в пуле. Поэтому часто люди плюют на эту головную боль и используют `packaged_task` и явные потоки.

Попытка дважды получить `future` или дважды поставить значение `promise` это исключение времени выполнения

```
1 std::promise<int> pr;
2 auto fut1 = pr.get_future();
3 auto fut2 = pr.get_future(); // Error: future already retrieved
4 pr.set_value(10);
5 pr.set_value(10); // Error: promise already satisfied
```

Если объект обещания у которого получено обещание, умирает раньше, чем выполняет его, это ошибка времени выполнения (нарушенное обещание).

```
1 promise<int> pr;
2 auto fut = pr.get_future();
3
4 {
5     promise<int> pr2(move(pr));
6 } // Error: broken promise
```

Разумное использование `future` и `promise` полностью лежит на плечах программиста. Объекты этих классов не могут быть скопированы (да и вряд ли их копирование несло бы полезную семантическую нагрузку), зато могут быть перемещены, в том числе возвращены из функции. Функция, которая возвращает `future` называется `future provider`. Я предпо читаю переводить это как “дающая обещание” потому что “поставщик будущего” звучит слишком мрачно и торжественно.

## 6.2 Конкуренция за данные

Жизнь была бы прекрасна, если бы каждую проблему удавалось распараллелить таким образом, чтобы потоки никогда не использовали общих ресурсов и вся система представляла собой набор изолированных процессов, максимум что делающих это обменивающихся сообщениями через опять-таки неконкурентные каналы вроде сокетов.

В реальных системах за пределами искусственных или крайне низковневых примеров, такое распараллеливание это сладкий сон. Жизнь, как водится, жёстче. Всегда есть некоторый разделяемый контекст, всегда есть борьба за него у работающих с ним потоков и всегда есть необходимость в синхронизации.

В этом разделе речь пойдёт в основном про так называемые примитивы синхронизации: мьютексы разных типов, условные переменные и т.д., а также про высокоуровневые методы работы с ними посредством так называемых защёлок (`lock_guard`, `unique_lock`).

Опять-таки тут есть некоторые языковые трудности. В русском языке блокировка это процесс. Использование слова “блокировка” для обозначения объекта блокировки, вроде того же `std::lock_guard` выглядит странно. Получается, что мы иногда блокируем, хм... объекты блокировки (*obtaining lock*). Поэтому без англицизмов вроде “взять” или “защёлкнуть” вряд ли получится обойтись.

Самое главное, для чего нужны блокировки это для того, чтобы избежать гонок (*races*, *race conditions*).

### 6.2.1 Гонки

Гонки проще всего объяснять через примеры. Мой любимый пример на гонку, это, собственно, гонка, когда простой счётчик сначала гонится вверх до большого значения а потом поворачивает и гонится вниз до нуля. Источник проблем в приведённом ниже коде, разумеется, в том, что счётчик глобальный, а в функцию могут одновременно зайти несколько потоков.

```
1 void __attribute__((noinline)) use(int c) {
2     asm("");
3 }
4
5 int x;
```

```

6
7 void race() {
8     // going up
9     for(int i = 0; i < 1000000; ++i) {
10         x += 1; use(x);
11     }
12
13     // going down
14     for(int i = 0; i < 1000000; ++i) {
15         x -= 1; use(x);
16     }
17 }
```

Можно обратить внимание, что эта функция не делает ничего такого: речь о простых сложениях и вычитаниях. Вызов пустой функции с пустой ассемблерной вставкой внутри там просто для того, чтобы код не соптимизировался слишком умным компилятором.

Очевидно, что если запустить один поток, работающий с этой функцией, то итогом будет ноль.

```

1 thread t{race};
2 t.join();
3 assert(x == 0);
```

Но что если попробовать два потока?

```

1 thread t1{race};
2 thread t2{race};
3 t1.join();
4 t2.join();
5 cout << x << endl;
```

В этом случае всё сложно. Попробуйте запустить эту программу несколько раз: вы осознаете, что её можно использовать как весьма достоверный датчик случайных чисел.

Почему так получается? Из-за борьбы (говорят также – гонки) потоков за ресурс, в данном случае глобальную переменную x. Потоки читают, меняют и перезаписывают эту переменную без малейшей договорённости друг с другом и постоянно лезут друг другу под руки. Такая ситуация это UB в языке C++ и это в частности означает, что может произойти или не произойти всё, что угодно.

Если заменить просто `int` в типе счётчика на `volatile int`, не меняется ровным счётом ничего: волатильность отключает оптимизации вокруг счётчика, но сами операции, которые его изменяют продолжают быть несинхронизированными по побочным эффектам.

Иногда гонки бывают не столь очевидны и даже умудряются много лет существовать в больших проектах годами, иногда удивляя пользователей странными трудноуловимыми багами. Чтобы их поймать, бывает необходимо использовать сложные инструменты вроде hellgrind.

Ситуация, как и было анонсировано, лечится синхронизацией разделяемых ресурсов.

### 6.2.2 Простая синхронизация

Простейшим способом синхронизации является мьютекс (mutex, от английского *mutually exclusive*, взаимно исключающий). Нормального перевода на русский язык тут нет (взаключ? взаиск?). Как следует из его английского названия, мьютекс предоставляет доступ к защищённой им части кода только одному потоку в каждый момент времени. Поток должен выйти из защищённого участка, прежде, чем туда пустят другой поток

```

1 int x;
2 mutex mforx;
3
4 void race() {
5     for(int i = 0; i < 1000000; ++i) {
6         mforx.lock();
7         x += 1; use(x);
8         mforx.unlock();
9     }
10
11    for(int i = 0; i < 1000000; ++i) {
12        mforx.lock();
13        x -= 1; use(x);
14        mforx.unlock();
15    }
16 }
```

Теперь результатом всегда будет ноль: вместо того, чтобы конфликтовать, потоки цивилизованно выстроются в очередь.

Ментальная модель мьютекса это именно очередь к ресурсу. Потоки подходят по одному, делают что надо и снова встают в очередь если надо сделать что-то ещё. Другое дело, что порядок подхода к кассе не определён, но обычно планировщики операционных систем достаточно честны.

**Вопрос к студентам:** осталось ли что-то случайное в рассматриваемой программе? Например, если мы замерим сколько раз значение `x` пересекло 10, получим ли мы детерминированный результат?

Стоит также обратить внимание как именно закрыты куски кода мьютексами: по факту закрыто ровно то, что нужно. Такие мьютексы называются тонко настроенными (fine-grained). И наоборот если закрыть всю функцию, это будет грубо настроенный (coarse-grained) мьютекс.

Конечно идея делать руками `lock()` и `unlock()` тут же вызывает желание написать обёртку и больше так не делать. Эта обёртка уже есть в стандарте и она называется `std::lock_guard`. Это очень обобщённая обёртка: она, грубо говоря, берёт что угодно у чего есть методы `lock` и `unlock` и вызывает `lock` в конструкторе и `unlock` в деструкторе.

```

1 int x;
2 mutex mforx;
3
4 void race() {
5     for(int i = 0; i < 1000000; ++i) {
6         lock_guard<mutex> lk{mforx};
7         x += 1; use(x);
8     } // unlocked
9
10    for(int i = 0; i < 1000000; ++i) {
11        lock_guard<mutex> lk{mforx};
12        x -= 1; use(x);
13    } // unlocked
14 }
```

Очень распространённая ошибка это не дать гарду имя.

```
1 lock_guard<mutex>(mforx); // probably WRONG
```

В этом случае всё будет хорошо, будет создан временный объект, который залочит мьютекс, доживёт до конца полного выражения и разложит мьютекс сразу же в том месте, где написана точка с запятой. Это, скорее всего, не то, что программист хочет, но это часто встречается.

Мьютексы не бесплатны в смысле производительности.

Например, они убивают возможность векторизации кода

```
1 for simd (i = 0; i < 64; ++i) {  
2     lock_guard<mutex> lk{m};  
3     A[i] = B[i] + C[i];  
4 }
```

Если здесь код будет векторизован, то следующая линия SIMD будет ждать мьютекс с предыдущей, который, разумеется, ещё не будет отпущен.

Кроме того, потоки всё-таки сериализуются на блокировках. За красивым словом “сериализуются” спрятано то, что они просто стоят в очереди и ничего не делают. Это несколько рушит производительность многопоточных систем.

Но обычно мы всё-таки готовы платить эту цену. К сожалению, часто этого оказывается недостаточно.

### 6.2.3 Проблемы интерфейсов

Тот факт, что вы обложились мьютексами, не гарантирует от проблем, заложенных при проектировании интерфейсов.

Рассмотрим буфер с обычными методами: `empty`, `push`, `pop` и `top`, подобный тому, который рассматривался при разговоре про исключения (см. 5.2.5)

```
1 template <typename T> struct MyBuffer {  
2     push(const T& x);  
3     void pop();  
4     T top() const;  
5     bool empty() const;  
6  
7     private:  
8         mutex bufmut_;  
9         // .... etc ....  
10    };
```

**Вопрос к студентам:** вы помните почему разделены `top` и `pop`?

Если не помните, перечитайте упомянутый выше раздел и освежите память.

Увы, многопоточность вносит дальнейшие корректизы в проектирование. С точки зрения многопоточности, самая лучшая и самая простая функция здесь это, конечно, `top`

```
1 template <typename T> T MyBuffer::top() const {
2     return arr_[used_ - 1];
3 }
```

Никаких блокировок и никакой синхронизации тут в принципе не нужно, так как честный `const` всегда означает thread safe. Сколько угодно потоков могут зайти в эту функцию и всё правильно прочитать.

Чуть сложнее функция `pop`, так как она всё-таки меняет инварианты класса и там придётся сделать мьютекс.

```
1 void pop () {
2     lock_guard<mutex> lk{bufmut};
3     used_ -= 1;
4     destroy(arr_ + used_);
5 }
```

Тем не менее, это не особо спасает. Даже если внутри этого класса каждый метод заведён под единый для объекта мьютекс, существует сразу два сценария, способных потенциально вызвать ошибку

Первый это гонка между `pop()` и `empty()`. Представим следующий кусок кода.

```
1 if (!s.empty()) {
2     auto elem = s.top();
3     s.pop();
4     use(elem);
5 }
```

Два потока заходят сюда и оба проходят проверку на непустоту, хотя на самом деле остался только один элемент. Далее первый поток пытается его снять (через `pop`) и второй тоже, получаем взрыв от разменования нулевого указателя (в лучшем случае `assert` внутри `pop`, если его туда поставить).

Да, это тоже гонка. Хотя с точки зрения модели исполнения C++ никакого race condition здесь не произошло и никакого UB (ну кроме разменования нулевого указателя) здесь нет. Но в логическом смысле это та же самая проблема: два потока одновременно вошли в один участок кода и разрушили его логику работы. Увы, никакой hellgrind такое не

поймает.

Второй возможный сценарий логической ошибки в таком классе это гонка между `top()` и `pop()` и вот это уже совсем мрачно. Два потока входят в тот же кусок кода когда в стеке есть два элемента. Далее один из них снимает первый и второй снимает первый. Далее оба удаляют каждый свой элемент. В итоге не происходит даже никакого взрыва, просто из буфера удалено два разных элемента, а у каждого из двух потоков две одинаковые копии.

Как можно пересмотреть интерфейс? При этом безопасность относительно исключений тоже хотелось бы сохранить

```

1 template <typename T> struct MyBuffer {
2     push(const T& x);
3
4     // This is not exception safe
5     // T pop();
6
7     // This is problematic in multi-threaded code
8     // void pop();
9     // T top() const;
10    // bool empty() const;
11
12    // pop if container not empty. Return true on success
13    bool try_pop(T& value);
14    bool empty() const;
15 private:
16     mutex bufmut_;
17     // .... etc ....
18 };

```

Из соображений возможной многопоточности хотелось бы максимально объединить извлечение из буфера и освобождение памяти в интерфейсе. Первая идея как этого добиться это извлекать по выходному параметру-ссылке и не более чем пытаться извлекать, возвращая в случае чего информацию о неудаче. Эта идея иллюстрирована в листинге выше

Ещё одна идея, приведённая в [16] это извлекать умный указатель. Тогда в случае неудачи он может быть `nullptr`.

```

1 shared_ptr<T> pop();

```

На самом деле, этот вопрос куда сложней, чем кажется. Все контейнеры стандартной библиотеки уже исторически спроектированы так, как будто многопоточных программ не существует. И это проблема, которая нам всем ещё много раз аукнется. Я имею в виду не в этих лекциях, а вообще.

#### 6.2.4 Экзотичные блокировки

Ещё одна частая проблема интерфейсов состоит в том, что попытка защёлкнуть уже защёлкнутый мьютекс это UB. Таким образом, если в некоем классе в начале каждого метода стоит защёлкивание мьютекса, один метод этого класса уже не может вызвать другой его метод.

Попытка решения может использовать класс `std::recursive_mutex` – специальный мьютекс с внутренним счётчиком, который считает количество защёлкиваний.

```

1 template <typename T> struct sometype {
2     foo() {
3         lock_guard<recursive_mutex> lk{mut_};
4         // other works
5     }
6     bar() {
7         lock_guard<recursive_mutex> lk{mut_};
8         foo();
9         // other work
10    }
11
12 private:
13     mutable recursive_mutex mut_;
14     // .... etc ....
15 };

```

Это работает, но профессионалы считают, что от такого кода дурно пахнет.

История появления рекурсивных мьютексов в POSIX и аргументация против них изложена, например, здесь: <http://www.zaval.org/resources/library/butenhof1.html>

Ещё один пример странной идиомы это `std::timed_mutex` – специальный мьютекс, который предоставляет целых два новаторских метода.

Во-первых он позволяет ждать себя не бесконечно, а с каким-то таймаутом

```

1 auto now=std::chrono::steady_clock::now();
2 res = test_mutex.try_lock_until(now + 10s);
3 if (!res) {
4     // we are tired of waiting here
5 }
```

Далее если результат `false`, то можно делать что угодно (например подождать ещё раз).

Во-вторых он позволяет заблокировать себя не более чем на какое-то время.

```

1 if (test_mutex.try_lock_for(Ms(100))) {
2     // we have 100 milliseconds here
3 }
```

Что делать в эти 100 миллисекунд каждый решает для себя сам. Возможно это будет что-то важное. Возможно кто-то просто создаёт таким образом хитрый плавающий случай гонки и гарантирует себе рабочее место на многие годы вперёд. Кто знает...

Разумеется антипаттерны не ходят по одиночке. Поэтому, в частности, `std::recursive_timed_mutex` тоже существует. Я позволю себе не комментировать его.

У мьютекса, ограниченного по времени есть только одно преимущество: он не подвержен взаимным блокировкам. И, кажется, пришла пора поговорить об этой, второй после гонок по важности, проблеме в борьбе за разделяемые ресурсы.

### 6.2.5 Взаимные блокировки

Слишком хорошо заблокированный код порождает взаимные блокировки мьютексов, которые, среди прочего, делятся на “мёртвые” блокировки (`deadlocks`) и “живые” блокировки (`livelocks`). О живых речь пойдёт чуть позже, сейчас в основном поговорим о мёртвых.

Допустим, нужно написать `swap` для уже рассмотренного выше класса `MyBuffer`. Первая идея как сделать это в потокобезопасном стиле это написать нечто вроде захвата обоих мьютексов:

```
1 template <typename T> void
```

```

1 MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
2     std::lock_guard<mutex> lk1{bufmut_};
3     std::lock_guard<mutex> lk2{rhs.bufmut_};
4     std::swap(data_, rhs.data_);
5     std::swap(size_, rhs.size_);
6     std::swap(used_, rhs.used_);
7 }
8 }
```

Увы, в этот момент у нас начинаются проблемы. Во первых этот метод теперь вовсе не noexcept, так как `lock_guard` вполне может бросить исключение. И во вторых он обязательно его бросит если `lhs == rhs` и метод пытается дважды захватить один и тот же мьютекс, что запрещено правилами.

Но даже игнорируя это (в конце концов, добавить проверку на неравенство несложно), этот код чудовищно плох.

Сценарий ошибки такой: первый поток меняет местами `x` и `y`. В это время второй меняет местами `y` и `x`. Первый захватил мьютекс для `x` и ждёт мьютекса для `y`. Второй в это время взял мьютекс для `y` и ждёт мьютекса для `x`. Собственно всё, типичная “мёртвая” блокировка случилась. Теперь её разрешения можно ждать до бесконечности.

Хотелось бы захватить оба одновременно и освободить в правильном порядке, избегая мёртвых блокировок. До C++17 это делалось странным образом.

```

1 template <typename T> void
2 MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
3     std::lock(bufmut_, rhs.bufmut_);
4     std::swap(data_, rhs.data_);
5     std::swap(size_, rhs.size_);
6     std::swap(used_, rhs.used_);
7     bufmut_.unlock();
8     rhs.bufmut_.unlock();
9 }
```

Алгоритм `std::lock` делает в implementation-defined манере любую грешную магию, чтобы избежать сценариев мёртвых блокировок. К сожалению, это не RAII класс, поэтому об освобождении надо заботиться явно.

Чтобы хоть как-то ввести это в рамки RAII, для `lock_guard` был введён параметр конструктора, позволяющий усыновить зашёлкнутый

кем-то другим замок.

```

1 template <typename T> void
2 MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
3     std::lock(bufmut_, rhs.bufmut_);
4     std::lock_guard<mutex> lk1{bufmut_, std::adopt_lock};
5     std::lock_guard<mutex> lk2{rhs.bufmut_, std::adopt_lock};
6     std::swap(data_, rhs.data_);
7     std::swap(size_, rhs.size_);
8     std::swap(used_, rhs.used_);
9 }
```

Что здесь происходит? Сначала вызывается `lock`, как и раньше. Потом зашлёнкнутые им в правильном порядке мьютексы захватываются в обёртки. Потом в конце обёртки умирают и в деструкторах мьютексы освобождаются.

Это было работоспособно, но очень уныло и континуитивно и в C++17 это наконец поправили с помощью класса `scoped_lock`

```

1 template <typename T> void
2 MyBuffer::swap(MyBuffer<T> &rhs) noexcept {
3     std::scoped_lock(bufmut_, rhs.bufmut_);
4     std::swap(data_, rhs.data_);
5     std::swap(size_, rhs.size_);
6     std::swap(used_, rhs.used_);
7 }
```

Если функция использует внутри себя более одного мьютекса и у вас достаточно свежий компилятор, всегда предпочтите `scoped_lock`.

Ещё одна функция для безопасной условной блокировки множественных мьютексов это `std::try_lock`. Она работает в принципе так же, как `std::lock`, только для неё так и не ввели `scoped` варианта, поэтому всё приходится делать по старинке.

Например пусть два потока независимо ведут разные счётчики

```

1 void increment_loop (int idx, const char *desc) {
2     for (int i = 0; i < 10; ++i) {
3         {
4             lock_guard<mutex> lock {mut[idx]};
5             ++counter[idx];
6         }
7         lock_guard<mutex> lcout {mcout};
```

```

8         cout << desc << ":" << counter[idx] << endl;
9     }
10    }
11    std::this_thread::sleep_for(100ms);
12 }
13 }
```

Тогда функция третьего потока который пытается по мере возможности выводить на экран сумму этих счётчиков, может выглядеть так:

```

1 void update_loop () {
2     while (overall_count < 20) {
3         int result = try_lock(mut[0], mut[1]);
4         if (result == -1) {
5             lock_guard<mutex> lock1 {mut[0], std::adopt_lock};
6             lock_guard<mutex> lock2 {mut[1], std::adopt_lock};
7             overall_count = counter[0] + counter[1];
8             {
9                 lock_guard<mutex> lcout {mcout};
10                cout << "overall: " << overall_count << endl;
11            }
12        }
13        std::this_thread::sleep_for(200ms);
14    }
15 }
```

В переменную `result`, алгоритм `std::try_lock` записывает `-1` если у него получилось взять все локи или номер (начиная с нуля) того, который не получилось.

Классическим примером взаимной блокировки является удаление из двусвязного списка. Допустим некто решил сделать список, у которого каждый узел защищен мьютексом

```

1 struct list<T>::node {
2     node *prev_;
3     node *next_;
4     mutex m_;
5     T data_;
6 };
```

Тогда при удалении элемента функция удаления должна блокировать предыдущий и следующий элемент, чтобы убедиться, что его указатели

не изменяется во время удаления. Это может выглядеть как-то так:

```

1 void erase(node *cur) {
2     assert(cur);
3     {
4         lock_guard<mutex> lk {cur->m_};
5         assert(cur->next_);
6         assert(cur->prev_);
7         lock_guard<mutex> lknex {cur->next_->m_};
8         lock_guard<mutex> lkprev {cur->prev_->m_};
9         cur->next_->prev_ = cur->prev_;
10        cur->prev_->next_ = cur->next_;
11    }
12    delete cur;
13 }
```

Код несколько упрощённый, так как предполагает что элемент в середине. Реальный код будет несколько сложней. Сценарий взаимной блокировки крайне простой: в списке из узлов A, B, C, D, первый поток удаляет B, а второй C. Первый берёт заблокировал B и ждёт разблокировки C, а второй наоборот, заблокировал C и ждёт разблокировки B. Приплыли.

### 6.2.6 Одноразовые события

Взаимные блокировки это не единственный источник проблем при работе с примитивами синхронизации. Типичная задача при организации работы с каким-либо ресурсом это его одноразовая инициализация

```

1 resource *resptr;
2 mutex resmut;
3
4 void foo() {
5     {
6         lock_guard<mutex> lk{resmut};
7         if (!resptr)
8             resptr = new Resource();
9     }
10    resptr->use();
11 }
```

Это работает, но это слишком консервативно. Одноразовое создание – вот что по настоящему надо заблокировать. Сериализовать потоки, выстраивая их в очередь на безопасной проверке коррекции, которая 99% времени работы будет тривиально выполнена это слишком жёстко. Увы, проверку нельзя просто так взять и вынести вверх.

**Вопрос к студентам:** какой сценарий ошибки при работе со следующим кодом?

```

1 void foo() {
2     if (!resptr) {
3         lock_guard<mutex> lk{resmut};
4         resptr = new resource();
5     }
6     resptr->use();
7 }
```

В попытках избежать проблем, люди часто находят себе их в большем количестве, используя паттерн, печально известный как double-checked lock, DCL.

```

1 if (!resptr) {
2     lock_guard<mutex> lk{resmut};
3     if (!resptr)
4         resptr = new resource();
5 }
6 resptr->use();
```

Увы ситуация стала хуже: теперь чтение снаружи блокировки не синхронизировано с изменением внутри блокировки, что может привести к действительно странным вещам: например поток может увидеть resptr в полуготовом состоянии, когда ему уже присвоено значение, но память для него ещё не выделена.

Отлаживать такие баги очень тяжело, так что проще их не делать, запомнив (по крайней мере по изучения атомиков), что DCL как паттерн безнадёжно плох.

Это мотивирует такой полезный класс как `std::once_flag`. Правильным подходом к инициализации в этом коде будет

```

1 resource *resptr;
2 std::once_flag resflag;
3
4 void init_resource() {
```

```

5     resptr = new resource();
6 }
7
8 void foo() {
9     std::call_once(resflag, init_resource);
10    resptr->use();
11 }
```

Использованная здесь функция `std::call_once`, кажется, понятна без пояснений.

Ещё один безнадёжно плохой паттерн это использование `volatile` переменных для того, что люди считают синхронизацией.

```

1 volatile int resready = 0;
2 resource *resptr;
3
4 // will be called by thread 1
5 void foo() {
6     resptr = new resource();
7     resready = 1;
8 }
9
10 // will be called by thread 2
11 void bar() {
12     while (!resready) {
13         std::this_thread::yield();
14     }
15     resptr->use();
16 }
```

Проблема ровно та же: отсутствие синхронизации между чтением в потоке 2 и записью в потоке 1 открывает портал в ад, куда программист с радостью въезжает на мёртвом осле.

К глубокому сожалению автора, такие схемы часто годами бессбойно работают на x86 из-за total store ordering. Но это ещё хуже, потому что рано или поздно всё переезжает на arm. Вернее пытается переехать и не доезжает.

Про относительность одновременности и про то как разные потоки могут видеть несинхронизированные события, будет больше далее (см. 6.3.3).

### 6.2.7 Условные переменные

Возможности блокировки `lock_guard`, ограниченной областью видимости, вызывают ту же критику, что и обычные scoped pointers: полученную в одной функции блокировку нельзя отдать в другую. Но кроме того, её ещё и нельзя на время освободить и потом опять защёлкнуть.

Обе эти проблемы решает класс `unique_lock`. Увы, он несёт с собой несколько больший оверхед, потому что наличие блокировки в его случае превращается во флаг, который надо хранить и проверять.

Наиболее частое применение `unique_lock` это работа с условными переменными. Допустим в коде ниже функция `preparation` запускается в отдельном потоке и готовит данные для обработки (что здесь эмулируется сном, но понятно, что там может быть любая нетривиальная работа). Разумеется, поскольку данные общие, для подготовки она должна взять блокировку. Вопрос в том как ей известить функцию `processing`, исполняющуюся в отдельном потоке, что данные готовы?

Механизм, который работает в примере ниже основан на использовании условной переменной `std::condition_variable`. Поток с помощью `unique_lock` берёт блокировку, но сразу по вызову `data_cond.wait(lk)` отдаёт её и переходит в режим ожидания, который кончается по сигналу `data_cond.notify_one()`, после чего поток обработки пробуждается и снова получает блокировку обратно. Сделать такую систему, не используя `unique_lock` было бы затруднительно.

```

1 mutex gmut;
2 condition_variable data_cond;
3 int counter;
4
5 // this implementation is buggy, see below
6 void processing() {
7     for (;;) {
8         unique_lock<mutex> lk{gmut};
9         data_cond.wait(lk);
10        // here lock for gmut obtained
11        cout << counter << endl;
12    }
13 }
14
15 void preparation() {
16     for(;;) {

```

```

17     std::lock_guard<std::mutex> lk{gmut};
18     // here lock for gmut obtained
19     counter += 1;
20     std::this_thread::sleep_for(100ms);
21     data_cond.notify_one();
22 }
23 }
```

К сожалению, механизм условных переменных не слишком надёжен, и, в частности, подвержен так называемым внезапным пробуждениям (spurious wakeups). Таким образом, код выше на самом деле довольно плох: при внезапном пробуждении внутри функции `processing` могут быть обработаны не готовые данные.

Можно использовать явный цикл с дополнительной переменной

```

1 unique_lock<mutex> lk{gmut};
2
3 // protecting from spurious wakeups
4 while(!data_ready) {
5     data_cond.wait(lk);
6 }
7
8 // here lock for gmut obtained
9 cout << counter << endl;
```

Ну и, соответственно выставлять её в `preparation`. В стандарте предусмотрено некоторое облегчение судьбы программиста: ожидание с критерием пробуждения

```

1 unique_lock<mutex> lk{gmut};
2
3 // protecting from spurious wakeups
4 data_cond.wait(lk, []{ return data_ready; });
5
6 // here lock for gmut obtained
7 cout << counter << endl;
```

Критерий пробуждения задаётся любым функтором, тут он довольно прост, так что можно лямбдой прямо на месте.

Почему вообще возникает такая странная вещь как внезапные пробуждения? В стандарт C++ они перекочевали из стандарта POSIX (как и большинство всего остального). Но откуда они в POSIX? Вдумчивое

гугление позволяет установить причины: в POSIX есть такая вещь, как сигналы. Ожидание условной переменной реализовано через системный вызов. Но любой блокирующий системный вызов возвращается когда процесс получает сигнал. Далее функция ожидания условной переменной могла бы снова зайти в тот же системный вызов и продолжить ожидание. Но увы, в этом случае настоещее пробуждение может быть пропущено: как раз за то небольшое время, которое нужно, чтобы снова войти в системный вызов. Единственный способ избежать этого – разрешить случайные пробуждения, переложив ответственность за их обработку на программиста.

### 6.2.8 Разделяемые блокировки

Очень частый на практике случай это редко обновляемый (но всё-таки обновляемый) ресурс. Например хорошо устоявшаяся база данных, из которой читают в миллион раз чаще, чем записывают что-то новое. Или структура данных вроде справочника.

До 2017-го года нормативного способа работы с такими ситуациями просто не было. Но в C++17 были стандартизированы классы `shared_mutex` и `shared_lock`. Отличие `shared_mutex` в том, что она предоставляет как разделяемую (через `shared_lock`) так и эксклюзивную (через `unique_lock`) блокировку. Все потоки, читающие разделяемый ресурс, могут брать разделяемую блокировку и читать в своё удовольствие, не толкаясь локтями. При этом каждое изменение требует уже эксклюзивной блокировки. Разумеется, пока удерживается эксклюзивная блокировка, брать разделяемую просто нельзя и наоборот.

```
1 std::shared_mutex m_;
2 T value_;
3
4 T get() const {
5     std::shared_lock<std::shared_mutex> lock(m_);
6     return value_;
7 }
8
9 void modify(const T &newval) {
10    std::unique_lock<std::shared_mutex> lock(m_);
11    value_ = newval;
12 }
```

В этом коде показано, как устроен доступ через `get` и `modify` к общему ресурсу, охраняемому разделяемым мьютексом.

Интересно провести аналогию с умными указателями. При движении по линии `scoped` → `unique` → `shared`, понятия всё дальше расходятся.

## 6.3 Атомарность

Обычные примитивы синхронизации, будучи правильно использованы, почти всегда работают хорошо. Но есть случаи, когда хочется большего.

В своём докладе [41], Саттер привёл интересную аналогию: если традиционный подход к исключительному доступу похож на установку на дорогах светофоров, то работа с атомиками и неблокирующими структурами данных, похожа на строительство многоровневых кольцевых развязок, когда автомобиль может менять направление движения, почти не снижая скорости и не рискуя разбиться о встречный поток.

Конечно, если такую развязку строит неопытный программист, она с грохотом рушится, увлекая за собой всё.

На самом деле, нельзя сказать, что в прошлых разделах речь совсем не шла об атомиках. Например, что такое мьютекс под капотом? Допустим, целевая архитектура, как это часто бывает, предусматривает всего две основных ассемблерных инструкции: exclusive load, позволяющую эксклюзивно загрузить пару байт и exclusive store, позволяющую эксклюзивно эту пару байт сохранить. Этих двух операций достаточно, чтобы заработал тип `atomic_flag`, представляющий минимальный атомарный интерфейс: `clear` и `test_and_set`. А этого уже достаточно чтобы написать простейший мьютекс

```

1 class my_mutex {
2     atomic_flag flag{ATOMIC_FLAG_INIT};
3 public:
4     void lock() { while (flag.test_and_set()); }
5     void unlock() { flag.clear(); }
6
7     my_mutex() = default;
8     my_mutex(const my_mutex &) = delete;
9     my_mutex & operator=(const my_mutex &) = delete;
10 };

```

Инициализация флага с помощью зарезервированного значения `ATOMIC_FLAG_INIT` обязательна, это нужно учитывать.

Техника постоянного ожидания, использованная в функции `lock` называется spinlock, циклическая блокировка (английский спинлок тоже встречается). Хороший спинлок каждую итерацию своего вращения отдаёт пару тактов операционной системе

```

1 void my_mutex::lock() {
2     while (flag.test_and_set()) {
3         std::this_thread::yield();
4     }
5 }
```

Такой мьютекс даже можно использовать с большинством локов, как это рассматривалось ранее.

Тип данных `atomic_flag` является уникальным в том смысле, что он является единственным, для которого гарантированно прямое (lock free) соответствие атомарным возможностям железа. Обычные атомики вполне могут быть внутри реализованы через подобный же спинлок.

### 6.3.1 Базовая атомарность

Ранее (см. 6.2.6) был рассмотрен печально известный паттерн DCL (double checked lock), о котором заслуженно утверждалось, что он совершенно сломан.

Тем не менее, использование атомиков позволяет его починить. Всё, что для этого нужно это сделать указатель на ресурс атомарным.

```

1 atomic<resource *> resptr { nullptr };
2 mutex resmut;
3
4 void foo() {
5     if (!resptr) {
6         lock_guard<mutex> lk{resmut};
7         if (!resptr)
8             resptr = new Resource();
9     }
10    resptr.load()->use(); // note .load()
11 }
```

Инициализации не было пока `resptr` был глобальным указателем и заполнялся нулями. Но сейчас инициализация важна, так как использованный шаблон класса `atomic<T>` по умолчанию оставляет в полях мусор как бы он ни был выделен.

Операции с атомиками обычно дороги, так что во фрагменте коде выше можно воспользоваться своего рода оптимизацией, предложенной Стефаном Лававеем

```

1 atomic<resource *> resptr { nullptr };
2 mutex resmut;
3
4 void foo() {
5     resource *p = resptr.load();
6     if (p == nullptr) {
7         lock_guard<mutex> lk{resmut};
8         if (resptr.load() == nullptr)
9             resptr = p = new resource();
10    }
11    p->use();
12 }
```

Здесь проверка и использование делаются относительно неатомарной локальной переменной, что делает вещи возможно несколько более быстрыми (и всё ещё безопасными).

Разумеется, я предполагаю, что вы уже знаете о возможностях `once_flag` и никогда не станете делать такие оптимизации.

Спорной, но интересной возможностью после 2011 года является возможность переписать этот код ещё проще (синглтон Майерса)

```

1 resource *getres() {
2     static resource *p = new resource(); // thread-local
3     return p;
4 }
5
6 void foo() {
7     resource *p = getres();
8     p->use();
9 }
```

Стандарт гарантирует, что инициализация всех статических переменных будет атомарной. Тёмная сторона у этого варианта заключается в том, что все статические переменные в однопоточных программах из-за него стали менее эффективными, потому что теперь в любую однопоточную программу такого вида, компилятор запихнёт ненужную многопоточную обвязку и синхронизацию.

На самом деле шаблон `atomic<T>` подходит для любого класса, но чаще всего используется для простых типов, вроде целых чисел и указателей. Чем более сложен тип, который заворачивается в этот шаблон, тем больше вероятность, что там внутри будет старый добрый спинлок,

а это убивает всю идею (почему бы тогда не пользоваться обычными примитивами синхронизации).

Есть несколько специализаций шаблона `atomic<T>`, который представляют разный интерфейс. Но общими являются функции `store` и `load`, которые позволяют атомарно загрузить и атомарно выгрузить значение атомика. Именно функция `load` используется выше для доступа к данным.

Самым главным и основополагающим у атомиков является метод, который часто сокращают до CAS (compare and swap) – обмен и установка значения. В классе `std::atomic<T>` он представлен в двух ликах: `compare_exchange_weak` и `compare_exchange_strong`.

Синтаксис вызова у них примерно одинаковый. В простейшем случае он выглядит вот так:

```
1 bool changed = x.compare_exchange_weak(expected, desired);
```

Этот вызов берёт `expected` по ссылке, а `desired` по значению. Он проверяет содержимое атомика `x` на равенство с `expected` и если да, то записывает новое значение равное `desired`, а в `expected` сохраняет старое значение. Иначе не делает ничего. Результат, разумеется, означает, прошла ли проверка. Слабые (weak) версии могут быть подвержены spurious failures, то есть вести себя так как будто проверка не прошла, хотя она прошла. Но зато на некоторых платформах они могут давать выигрыш в производительности.

Для специализаций атомиков для арифметических типов к этой функциональности добавляется сахар вроде `fetch_add` или `fetch_sub`.

При первом знакомстве с ними, атомики кажутся низкоуровневым, но простым и приятным инструментом. Нет заблуждения хуже. Ко всем проблемам классической синхронизации, атомики добавляют кучу проблем, специфичных собственно для атомиков. В частности это касается и взаимных блокировок.

### 6.3.2 Взаимные блокировки

Интересной разновидностью взаимных блокировок являются живые блокировки или лайвлоки (livelocks). При этом два потока продолжают неуклюже ворочаться и что-то делать, но постоянно лезут друг другу под руку и поэтому топчутся на месте. Они довольно сложны с обычными примитивами синхронизации, но обычны в случае с атомиками.

Предположим, существует некий важный ресурс, способный отслеживать своего владельца

```

1 struct User;
2
3 struct Resource {
4     atomic<User *> owner {nullptr};
5     void use();
6 };

```

Далее класс владельца реализует метод, позволяющий (в задумке) избежать мёртвой блокировки со вторым владельцем, тоже претендующим (возможно) на этот ресурс.

```

1 struct User {
2     atomic<bool> done {false};
3     void useItPolitely(Resource &res, User &other) {
4         while (!done) {
5             // if owner not me, wait
6             if (res.owner != this) {
7                 std::this_thread::yield();
8                 continue;
9             }
10
11             // if owner me, but other not done, yield
12             if (!other.done) {
13                 res.owner = &other;
14                 continue;
15             }
16
17             res.use();
18         }
19     }
20 };

```

Ну и наконец вызывающая программа создаёт оба ресурса и пускает их в двух потоках.

```

1 Resource common;
2 User fst, snd;
3 common.owner = &fst;
4 thread t1{[&]() mutable { fst.useItPolitely(common, snd); }};
5 thread t2{[&]() mutable { snd.useItPolitely(common, fst); }};

```

```
6 t1.join(); t2.join();
```

В принципе ни в какой момент времени тут никто не заблокирован. Наоборот, оба потока работают достаточно интенсивно, постоянно уступая друг другу ресурс. Никакого прока в этом, в общем, нет.

Этот простой пример показывает, что атомики сложнее, чем кажется. И это действительно так. Однако, прежде, чем говорить о по-настоящему сложных вещах, необходимо ввести ещё одну концепцию: концепцию моделей памяти.

### 6.3.3 Модели памяти и теория относительности

Точно так же как и с базовой синхронизацией, можно начать с простой игры. Игра называется *chasing counters* и выглядит совершенно невинно.

Есть два целых числа и два потока. Первый поток их увеличивает, второй проверяет, что одно больше другого

```
1 volatile long long a = 0, b = 0;
2
3 // a is always > b
4 void chase() {
5     for(;;) {
6         a++;
7         b++;
8     }
9 }
10
11 // check that a > b
12 void check() {
13     for(;;) {
14         long long bval = b;
15         long long aval = a;
16         if (aval < bval)
17             cout << "Wrong: " << aval << " " << bval << endl;
18     }
19 }
```

Запустив эту программу на архитектуре x86 вы, скорее всего, ничего не увидите на экране. Работает пресловутый total store ordering (из-за

чего многие и любят `volatile`, работает же). Но запустите это на ARM и вы можете получить совершенно удивительный результат: на экран выведется Wrong, причём не один раз.

Почему это может произойти? Дело в том, что одновременность событий в языке C++ относительна.

Базовым для последовательных событий в пределах одного потока является соотношение линейной упорядоченности sequenced before / sequenced after (то, что ранее называлось точками следования). Так например в следующем коде

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

Не определено, что выполнится раньше инкремент `x` или `y`. Тут компилятор свободен переупорядочивать действия как ему вздумается. Но в коде

```
1 int s1 = x++;
2 int s2 = y++;
3 int s = s1 / s2;
```

Порядок событий жёстко определён. Мы говорим, что `x++` здесь **упорядоченно раньше**, чем `y++`. Если эти переменные глобальные, разумеется. Если нет, никаких событий (side effects) тут просто нет.

Такие отношения задают в языке то, что называется SCO – sequential consistency ordering. Но компилятор это не единственное существо, которое может переупорядочивать код. Переупорядочением занимается процессор, у которого есть свой execution ordering и подсистема памяти, у которой есть свой observable effects ordering. Если много потоков, то вмешивается ещё и распределение тактов операционной системой.

В случае многопоточной программы, стандарт никак не регламентирует в каком порядке один поток увидит побочные эффекты, случающиеся в другом потоке, если только между ними нет соотношения **синхронизированности** (synchronized with). Синхронизированность создают базовые примитивы синхронизации (мьютекс вполне подойдёт) но на базовом уровне синхронизированность создают атомики, работающие в определённых моделях памяти.

Модели памяти это нечто, о чём раньше речь никогда не шла. Например, когда говорилось о CAS, я предпочёл просто показать синтаксис вызова.

```
1 bool changed = x.compare_exchange_weak(expected, desired);
```

Теперь можно приоткрыть занавесочку. Настоящее объявление этой функции внутри шаблона `std::atomic<T>` выглядит так:

```

1 bool compare_exchange_weak(T& expected, T desired,
2   memory_order success, memory_order failure) noexcept;
3
4 bool compare_exchange_weak(T& expected, T desired,
5   memory_order order = memory_order_seq_cst) noexcept;
```

Здесь показаны две обычные перегрузки. Есть ещё `volatile` перегрузки. Как видно, реально, с учётом аргумента по умолчанию синтаксис вызова выше выглядит вот так:

```

1 bool changed = x.compare_exchange_weak(expected, desired,
2   std::memory_order_seq_cst);
```

В каждой операции над атомиками, в том числе в обычной записи туда данных и чтении оттуда данных есть эта незримая добавочка: аргументы задающие модель памяти, с каким-то разумным умолчанием.

Следуя превосходному докладу [47] модели памяти проще всего объяснить на примере синхронизации простого кода между потоками.

Пусть есть две функции. В первой происходит установка целочисленной переменной и выставление атомарного флага, а во второй проверка атомарного флага и вывод переменной на экран.

```

1 int data;
2 atomic<bool> done {false};
3 memory_order mosrc = /* some memory order */
4 memory_order modst = /* some memory order */
5
6 // thread 1
7 void foo() {
8   data = 42;
9   done.store(true, mosrc);
10 }
11
12 // thread 2
13 void bar() {
14   while (!done.load(modst)) {
15     std::this_thread::yield();
16   }
17   cout << data << endl;
```

18 }

Я сознательно не указал конкретное значение `mosrc` и `modst`, потому что сейчас мы попробуем разные. Я буду ссылаться на этот пример как на барьер Бирбахера.

Всего моделей памяти шесть и все они перечислены в хедере `<atomic>`. Этот список не расширяем пользователем, изменить его может только стандартный комитет при очередном пересмотре стандарта.

```
1 namespace std {
2     enum memory_order {
3         memory_order_relaxed,
4         memory_order_consume,
5         memory_order_acquire,
6         memory_order_release,
7         memory_order_acq_rel,
8         memory_order_seq_cst
9     };
10 }
```

Три основных семантических категории моделей памяти, о которых следует знать это `relaxed`, `acquire-release` и `sequential`.

Самая слабая из них это `relaxed` семантика и соответствующая ей модель `std::memory_order_relaxed`. Если в барьере Бирбахера установить такой порядок упорядочения, то вывод на экран выведет что угодно, поскольку `relaxed` order не устанавливает `synchronized with` отношения, а только гарантирует атомарность самого атомика. Если речь только о передаче данных внутри атомика, то всё хорошо, но если нет, то увы.

**Вопрос к студентам:** что насчёт `chasing counters` с `relaxed` моделью (т.е. мы переделываем оба счётчика из `volatile` в `atomic` и работаем с ними `relaxed` семантикой)?

Достаточно сильная и во многом недооценённая модель это `acquire-release`. Если установить в примере с барьером

```
1 memory_order mosrc = memory_order_release;
2 memory_order modst = memory_order_acquire;
```

Или обе в одинаковое значение `memory_order_acq_rel`, то это синхронизирует все эффекты до `release` со всеми эффектами после `acquire`.

Одновременное присутствие синхронизированности и последования в стандарте называется временнOй зависимостью (`happens before`). Теперь

запись 42 случается всегда до вывода 42 на экран. Никакие записи не могут быть переупорядочены через releasing store вниз и никакое чтение не может быть переупорядочено через acquiring load вверх.

У release-acquire семантики есть уродливый брат-близнец, который официально не рекомендован к употреблению в стандарте. Это release-consume. Если установить

```
1 memory_order mosrc = memory_order_release;
2 memory_order modst = memory_order_consume;
```

То семантика меняется: теперь чтения могут быть переупорядочены вверх через consuming load только в том случае если они не зависимы. В данном случае зависимости нет, так что барьер Бирбахера перестаёт работать.

Когда вы пишете свой код, вы можете считать, что модель `memory_order_consume` просто не существует.

Разумеется у acquire-release есть проблемы, которые выявляются, если несколько усложнить барьер.

```
1 int data1, data2;
2 atomic<int> done {0};
3
4 // thread 1
5 void foo() {
6     data1 = 42;
7     done.store(1, memory_order_acquire);
8 }
9
10 // thread 3
11 void buz() {
12     data = 42;
13     done.store(2, memory_order_acquire);
14 }
15
16 // thread 2
17 void bar() {
18     while (1 != done.load(memory_order_release)) {
19         std::this_thread::yield();
20     }
21     cout << data1 << " " << data2 << endl;
22 }
```

Здесь закопан случай неопределенного поведения. При acquire-release если из атомика была считана единица, мы можем надеяться только на то, что учтены побочные эффекты из той функции, где эта единица была выставлена и не более того.

Третья и самая сильная семантическая категория это sequential consistent модель памяти `memory_order_seq_cst`. Увы она не помогает в рассмотренном выше случае, зато она помогает при работе с несколькими атомиками.

```
1 atomic<int> x{0}, y{0};  
2 memory_order mo = /* some memory order */  
3 memory_order como = /* corresponding to mo */  
4  
5 void foo() {  
6     x.store(1, mo);  
7     cout << y.load(como) << endl;  
8 }  
9  
10 void bar() {  
11     y.store(1, mo);  
12     cout << x.load(como) << endl;  
13 }
```

Этот пример при использовании acq-rel семантики способен выдать на экран оба нуля!

Ментальная модель здесь такая: у каждого атомика есть total order, но один атомик всё равно может видеть побочные эффекты другого в общем случае в неупорядоченном виде.

Вся эта вольница прекращается когда приходит sequential consistency. В этом случае этот пример всегда напечатает обе единицы. Эта модель памяти всегда устанавливает полный последовательный порядок между всеми атомиками в программе. Именно она является наиболее ограничивающей, наиболее мешает оптимизациям, и, удивительно, но именно эта модель является моделью по умолчанию. Для языка C++ это очень странно. Мы-то привыкли, что по умолчанию он стреляет в ногу разрывными.

### 6.3.4 Хрупкость атомарности

Я понимаю, что после экскурса в модели памяти, об атомиках может укрепиться негативное впечатление. Увы, на самом деле атомики **ещё** сложнее. В качестве короткой рекомендации начинающему программисту можно посоветовать просто никогда не использовать атомики.

Как обычно, более тонкие проблемы начинаются при проектировании классов и интерфейсов.

Ричард Пауэлл в своём докладе [46] приводит пример условного класса `DataHolder` с одним недетерминированно долгим методом, вычисляющим контрольную сумму.

```

1 class DataHolder {
2     int calc_checksum() const; // may be long
3 public:
4     int get_checksum() const {
5         return calc_checksum();
6     }
7 };

```

В таком виде этот класс не содержит неконстантных методов и тривиально безопасен для многопоточных программ. Но что если туда хочется добавить кэширование контрольной суммы? Это вполне логичное желание: вычисление может быть долгим, вызов частым и так далее.

```

1 class DataHolder {
2     mutable atomic<int> cached_checksum_ = 0;
3     mutable atomic<bool> cached_ = false;
4
5     int calc_checksum() const; // may be long
6
7 public:
8     DataHolder() = default;
9     DataHolder(const DataHolder &rhs) :
10         cached_checksum_(rhs.cached_checksum_.load()),
11         cached_(rhs.cached_.load()) {
12     }
13
14     int get_checksum() const {
15         if (!cached_) {
16             cached_ = true;

```

```

17     cached_checksum_ = calc_checksum();
18 }
19 return cached_checksum_;
20 }
21 };

```

Теперь для безопасности потоков здесь использованы атомики (не брать же всё под мьютекс) и вынужденно добавлен конструктор копирования, потому что атомики убивают копирование по умолчанию.

Это кажется удивительным, но в этом простом и очевидном коде допущены две грубые ошибки, каждая из которых может привести к тонким и сложно обнаружимым багам при работе программы.

Первая ошибка связана с порядком проверки условий внутри `get_checksum`

```

1 if (!cached_) {
2     cached_ = true;
3     cached_checksum_ = calc_checksum();
4 }

```

Сценарий ошибки такой: первый поток заходит в функцию и видит что `cached_ == false`. Он выставляет `cached_ = true` и тут в функцию входит второй поток, который видит, что `cached_ == true` и возвращает кэшированное значение, которое первый поток ещё не вычислил. Упс.

**Вопрос к студентам:** к счастью эта ошибка решается тривиальным изменением кода. Каким?

Вторая ошибка гораздо более удивительна: она связана с тем, что один поток может быть внутри `get_checksum`, а второй осуществлять чтение того же объекта для создания его копии в конструкторе копирования.

Сценарий ошибки такой: первый поток начинает копировать объект с неинициализированной контрольной суммой и выставляет её. Далее второй поток вычисляет контрольную сумму, устанавливает `cached_ = true` и тут просыпается первый поток и записывает в копию `cached_`. Упс. Теперь в копии есть и мусорная контрольная сумма и флаг что она корректна.

**Вопрос к студентам:** к счастью эта ошибка тоже решается тривиальным изменением кода. Каким?

Эта ошибка – тоже своего рода race condition, только гонка тут слу-

чается не за одну переменную а за пару семантически связанных переменных.

# Список иллюстраций

|      |                                                        |     |
|------|--------------------------------------------------------|-----|
| 2.1  | Модульная структура программы . . . . .                | 34  |
| 2.2  | Визуальное представление указателей . . . . .          | 55  |
| 2.3  | Визуальное представление массивов . . . . .            | 59  |
| 2.4  | Массивы в языке С . . . . .                            | 64  |
| 2.5  | Два способа выравнивания . . . . .                     | 103 |
| 3.1  | Ошибка двойного освобождения . . . . .                 | 142 |
| 3.2  | Идентичность и перемещаемость . . . . .                | 182 |
| 3.3  | Открытое наследование интерфейса . . . . .             | 186 |
| 3.4  | Открытый интерфейс как общий разъём для подключения    | 187 |
| 3.5  | Открытое наследование реализации . . . . .             | 200 |
| 3.6  | Способ делать конкретными классами только листья . . . | 201 |
| 3.7  | Иерархии классов и объектов . . . . .                  | 210 |
| 3.8  | Виртуальные функции при множественном наследовании .   | 210 |
| 3.9  | Ромбовидная схема . . . . .                            | 212 |
| 3.10 | Виртуальное наследование . . . . .                     | 213 |
| 3.11 | Сложная иерархия . . . . .                             | 219 |
| 3.12 | Паттерн Bridge . . . . .                               | 235 |
| 4.1  | Ленивые и энергичные вычисления . . . . .              | 332 |
| 5.1  | Принципиальное устройство строки . . . . .             | 378 |
| 5.2  | Поиск в строках . . . . .                              | 379 |

## *СПИСОК ИЛЛЮСТРАЦИЙ*

|      |                                                                                   |     |
|------|-----------------------------------------------------------------------------------|-----|
| 5.3  | Принципиальное устройство string view . . . . .                                   | 382 |
| 5.4  | Идея copy on write строк . . . . .                                                | 385 |
| 5.5  | Реализация copy on write строк в libstdc++ v5 . . . . .                           | 386 |
| 5.6  | Идея для small string optimizations: настоящий масштаб для схемы строки . . . . . | 387 |
| 5.7  | SSO строка в libgcc версии выше пятой . . . . .                                   | 388 |
| 5.8  | Stack Unwinding . . . . .                                                         | 393 |
| 5.9  | Иерархия стандартных исключений . . . . .                                         | 396 |
| 5.10 | Нарушение нейтральности . . . . .                                                 | 398 |
| 5.11 | Диаграмма классов двухуровневого буфера . . . . .                                 | 406 |
| 5.12 | Разделяемые указатели: контрольный блок . . . . .                                 | 423 |
| 5.13 | Устройство разделяемого указателя после make_shared . .                           | 424 |
| 5.14 | Алиасинг в разделяемых указателях . . . . .                                       | 425 |
| 5.15 | Слабые указатели и устройство контрольного блока . . . . .                        | 428 |
| 5.16 | Проблема make_shared и слабых ссылок . . . . .                                    | 429 |
| 5.17 | Первое представление об итераторах . . . . .                                      | 444 |
| 5.18 | Сплайс для односвязных списков . . . . .                                          | 447 |
| 5.19 | Адаптеры стека и очереди . . . . .                                                | 450 |
| 5.20 | Диаграмма преобразований для указателей . . . . .                                 | 483 |
| 5.21 | Диаграмма преобразований для итераторов . . . . .                                 | 484 |
| 5.22 | Прямое и обратное итерирование . . . . .                                          | 485 |
| 5.23 | Идиома erase-remove . . . . .                                                     | 497 |
| 5.24 | Группа перестановок S(3) . . . . .                                                | 504 |
| 5.25 | Орбита элемента 1 в S(3) . . . . .                                                | 505 |
| 5.26 | Обход группы и вычисление орбиты . . . . .                                        | 505 |
| 5.27 | Помещение в вектор двух строк с локальными аллокаторами                           | 532 |
| 5.28 | Иллюстрация проблем локальных аллокаторов . . . . .                               | 533 |
| 5.29 | Альтернативный односвязный список . . . . .                                       | 539 |

# Литература

- [1] Brian W. Kernighan, Dennis M. Ritchie, «*The C Programming Language, 2nd edition*», Prentice Hall, 1988
- [2] Peter van der Linden, «*Expert C Programming: Deep C Secrets*», Prentice Hall, 1994
- [3] Bjarne Stroustrup, «*The C++ Programming Language, The Special Edition*», Addison-Wesley, 2001
- [4] Bjarne Stroustrup, «*The C++ Programming Language (4th Edition)*», Addison-Wesley, 2013
- [5] Scott Meyers, «*Effective C++ : 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*», Addison-Wesley, 2005
- [6] Scott Meyers, «*Effective STL*», Addison Wesley, 2001
- [7] Scott Meyers, «*Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*», O'Reilly Media, 2014
- [8] Alexander Stepanov, Paul McJones, «*Elements of Programming*», Addison-Wesley, 2009
- [9] Nicolai M. Josuttis, «*The C++ Standard Library – A Tutorial and Reference*», Addison-Wesley, 1999
- [10] Nicolai M. Josuttis, «*The C++ Standard Library – A Tutorial and Reference, second edition*», Addison-Wesley, 2012
- [11] Davide Vandevoorde, Nicolai M. Josuttis, «*C++ Templates. The Complete Guide*», Pearson Education, 2003
- [12] Grady Booch, «*Object-oriented analysis and design with applications, third edition*», Addison-Wesley, 2007

## ЛИТЕРАТУРА

- [13] Herb Sutter, «*Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*», Addison-Wesley, 2000
- [14] Herb Sutter, «*More exceptional C++: 40 new engineering puzzles, programming problems, and solutions*», Addison-Wesley, 2002
- [15] Andrei Alexandrescu, «*Modern C++ Design. Generic programming and design patterns applied*», Addison-Wesley, 2001
- [16] Anthony Williams, «*C++ Concurrency in Action*», Manning, 2012
- [17] Steve McConnell «*Code Complete, Second Edition*», Microsoft Press, 2004
- [18] Herb Sutter and Andrei Alexandrescu «*C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*», Addison-Wesley Professional, 2004
- [19] ISO/IEC, «*International Standard, Programming Languages*», ISO/IEC 14882:1998, 1998
- [20] ISO/IEC, «*International Standard, Programming Languages*», ISO/IEC 9899:1990, 1990
- [21] ISO/IEC, «*International Standard, Programming Languages*», ISO/IEC 9899:1999, 1999
- [22] ISO/IEC, «*International Standard, Programming Languages*», ISO/IEC 9899:2011, 2011
- [23] ISO/IEC, «*International Standard, Programming Languages*», ISO/IEC 9899:2014, 2014
- [24] ISO/IEC, «*International Standard, Programming Languages*», ISO/IEC 14882:2011, 2011
- [25] ISO/IEC, «*International Standard, Programming Languages*», ISO/IEC 14882:2014, 2014
- [26] Barbara Liskov, «*Data Abstraction and Hierarchy*», Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications, 1987
- [27] Robert Martin, «*The Interface Segregation Principle*»

## ЛИТЕРАТУРА

- [28] Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John, «*Design Patterns: Elements of Reusable Object-oriented Software*», Addison-Wesley, 1995
- [29] Tom Cargill, «*Exception handling: a false sense of security*», C++Report, 1994
- [30] David Abrahams, «*Exception-safety in generic components*», 1998
- [31] Nicholas Ormrod, «*The strange details of std::string at Facebook*», CppCon, 2016
- [32] Mark Zeren, «*Rethinking strings*», C++Now, 2017
- [33] Антон Полухин, «*Как делать не надо. Велосипедостроение для профессионалов*», C++ Russia User Group, 2016
- [34] Jon Kalb, «*Exception Safe code (3 parts)*», CppCon, 2014
- [35] Niall Douglas, «*Mongrel Monads*», ACCU, 2017
- [36] Sean Parent, C++ Seasoning, GoingNative'2013
- [37] Sean Parent, Better Code: Data Structures, CppCon'2015
- [38] Alisdar Meredith, «*Making allocators work*», CppCon, 2014, parts 1 and 2
- [39] Andrei Alexandrescu, «*std::allocator Is to Allocation what std::vector Is to Vexation*», CppCon, 2015
- [40] Bob Steagall, «*How to write a custom allocator*», CppCon, 2017
- [41] Herb Sutter, «*Lock-Free Programming (or, Juggling Razor Blades)*», CppCon, 2014
- [42] Pablo Halpern, «*Allocators, the good parts*», CppCon, 2017
- [43] Pablo Halpern, «*Overview of parallel programming in C++*», CppCon, 2014
- [44] John Lakos, «*Local (arena) memory allocators*», CppCon, 2017, parts 1 and 2
- [45] David Sankel, «*C++17 std::pmr comes with a cost*», C++Now, 2018
- [46] Richard Powell, «*The Importance of Being const*», CppCon, 2015

## ЛИТЕРАТУРА

- [47] Frank Birbacher, «*Atomic's memory orders, what for?*», ACCU, 2017
- [48] Arne Mertz, «*Modern C++ features – keyword noexcept*», blog post, 2016
- [49] Howard Hinnant, «*stack\_alloc*», github, 2015

# Лицензия

Этот текст написан и распространяется под условиями Creative Commons CC BY-NC-SA License. Даная лицензия позволяет другим людям редактировать, поправлять и брать этот текст за основу для производных в некоммерческих целях при условии, что они указывают моё авторство и лицензируют свои произведения, производные от данного, на тех же условиях.

Условия: <http://creativecommons.org/licenses/by-nc-sa/3.0/deed.ru>

Текст: <http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode>

# Предметный указатель

- Argument dependent lookup, 96
- Array of pointers, 35
- assignment operator, 140
- Barton-Nackman trick, 281
- Binding, 322
  - C, 14
  - C declarations, 36
  - C++, 14
  - cast, 85
  - Cfront, 14
  - class template, 257
  - COAP, 416
  - Concepts, 543
  - Conforming, 18
  - const, 35
  - const\_cast, 85
  - Constraints, 548
  - constructor, 126
  - copy constructor, 140
  - Covariant, 439
  - CRTP, 282
  - Currying, 310
  - Cutting, 199
  - cv-qualifier, 36
  - dangling reference, 68
  - Decay, 59
  - Decay function, 116
  - declaration, 47
  - deep copy, 141
  - default arguments, 192
  - definition, 47
  - delete, 83
  - Demeter Law, 231
  - destructor, 126
  - DIP, 231
  - disambiguation, 266
  - Drill-Down behaviour, 152
  - Dynamic type, 188
  - dynamic\_cast, 214
  - emplace, 296
  - Friend, 222
  - Function Overloading, 71
  - function template, 243
  - Functor, 150
  - header, 34
  - Higher-order function, 310
  - Implementation defined, 18
  - implicit type cast, 128
  - Inheritance, 183
  - Interface inheritance, 183
  - ISO, 15
  - ISO standard, 15
  - ISP, 230
  - Kenig search, 97

## *ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ*

- library, 34
- LSP, 229
- lvalue, 52
- member function, 78
- Metaprogramming, 350
- multidimensional arrays, 63
- Name Mangling, 71
- namespaces, 93
- new, 83
- NPOD, 125
- NVI, 201
- OCP, 228
- ODR, 48
- operator, 146
- overload, 146
- overloading resolution, 73
- Parameter pack, 291
- partial specialization, 260
- placement new, 158
- POD, 125
- Point of Declaration, 48
- Program Structure, 33
- RAII, 139
- Range-based for, 473
- references, 66
- reinterpret\_cast, 85
- RTTI, 214
- rvalue, 52
- RVO, 144
- Sanitizers, 23
- Scope, 49
- Separation model, 331
- SFINAE, 329
- shallow copy, 141
- slice, 452
- SOLID, 227
- splice, 447
- SRP, 227
- Static Analysis, 24
- static members, 131
- static\_cast, 85
- strict aliasing, 56
- Strictly conforming, 18
- Stroustrup, 14
- Syntax violation, 18
- Template Polymorphism, 252
- template specialization, 258
- template template parameters, 286
- this, 78
- traits, 389
- Turing completeness of templates, 356
- Type inference, 107
- Type traits, 340
- type\_info, 216
- typedef, 37
- typename, 266
- Undefined behavior, 18
- Unspecified, 18
- Variadic Templates, 290
- vector capacity, 437
- volatile, 35