

30 главных
правил чистого,
безопасного
и быстрого
кода

КРАСИВЫЙ C++

Создатель языка C++
Бьери Страуструп
рекомендует



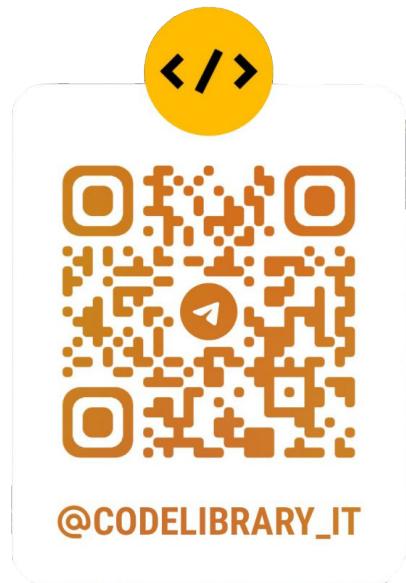
ДЖ. ГАЙ ДЭВИДСОН / КЕЙТ ГРЕГОРИ



Beautiful C++

30 Core Guidelines for Writing Clean, Safe,
and Fast Code

J. Guy Davidson
Kate Gregory



◆ Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo



Красивый C++

30 главных правил чистого, безопасного
и быстрого кода

Дж. Гай Дэвидсон

Кейт Грегори



Санкт-Петербург · Москва · Минск

2023

ББК 32.973.2-018.1
УДК 004.43
Д94

Дэвидсон Дж. Гай, Грегори Кейт

Д94 Красивый C++: 30 главных правил чистого, безопасного и быстрого кода. — СПб.: Питер, 2023. — 368 с.: ил. — (Серия «Для профессионалов»).
ISBN 978-5-4461-2272-1

Написание качественного кода на C++ не должно быть трудной задачей. Если разработчик будет следовать рекомендациям, приведенным в C++ Core Guidelines, то он будет писать исключительно надежные, эффективные и прекрасно работающие программы на C++. Но руководство настолько переполнено советами, что порой трудно понять, с чего начать. Начните с «Красивого C++»!

Опытные программисты Гай Дэвидсон и Кейт Грегори выбрали 30 основных рекомендаций, которые посчитали особенно ценными, и дают подробные практические советы, которые помогут улучшить ваш стиль разработки на C++. Для удобства книга структурирована в точном соответствии с официальным веб-сайтом C++ Core Guidelines.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с Pearson Education Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-0137647842 англ.
ISBN 978-5-4461-2272-1

© 2022 Pearson Education, Inc.
© Перевод на русский язык ООО «Прогресс книга», 2023
© Издание на русском языке, оформление ООО «Прогресс книга»,
2023
© Серия «Для профессионалов», 2023

Оглавление

https://t.me/it_boooks

| | |
|-------------------------------------|----|
| Избранные рекомендации по C++ | 14 |
| Предисловие | 17 |
| Вступление | 18 |
| О книге..... | 20 |
| Код примеров..... | 23 |
| Благодарности..... | 24 |
| Об авторах | 26 |
| От издательства | 28 |

ЧАСТЬ I BIKESHEDDING — ЭТО ПЛОХО

| | |
|--|----|
| Глава 1.1. Р.2. Придерживайтесь стандарта ISO C++ | 30 |
| Что такое стандарт ISO C++ | 30 |
| История C++ | 30 |
| Инкапсуляция вариаций | 32 |
| Вариации в окружении времени выполнения | 32 |
| Вариации на уровне языка C++ и компилятора | 33 |
| Расширения для C++..... | 34 |
| Защита заголовочных файлов | 35 |
| Вариации в основных типах | 35 |
| Нормативные ограничения | 36 |
| Изучение старых способов | 37 |
| Обратная совместимость в C++..... | 37 |
| Прямая совместимость и Y2K..... | 38 |
| Следите за последними изменениями в стандарте | 39 |
| IsoCpp..... | 39 |
| Конференции..... | 40 |
| Другие источники..... | 40 |

| | |
|--|----|
| Глава 1.2. F.51. Если есть выбор, используйте аргументы по умолчанию вместо перегрузки..... | 42 |
| Введение..... | 42 |
| Доработка ваших абстракций: дополнительные аргументы или перегрузка? | 43 |
| Тонкости разрешения перегрузки | 45 |
| Вернемся к примеру | 47 |
| Однозначная природа аргументов по умолчанию..... | 49 |
| Альтернативы перегрузке | 50 |
| Иногда без перегрузки не обойтись..... | 51 |
| Подведем итог | 52 |
| Глава 1.3. C.45. Не определяйте конструктор по умолчанию, который просто инициализирует переменные-члены; для этой цели лучше использовать внутриклассовые инициализаторы членов | 53 |
| Зачем нужны конструкторы по умолчанию | 53 |
| Как инициализируются переменные-члены..... | 55 |
| Что может случиться, если поддерживать класс будут два человека | 58 |
| Сборная солянка из конструкторов | 58 |
| Аргументы по умолчанию могут запутать ситуацию в перегруженных функциях..... | 60 |
| Подведем итог | 60 |
| Глава 1.4. C.131. Избегайте тривиальных геттеров и сеттеров | 62 |
| Архаичная идиома | 62 |
| Абстракции..... | 63 |
| Простая инкапсуляция..... | 66 |
| Инварианты класса..... | 69 |
| Существительные и глаголы..... | 71 |
| Подведем итог | 72 |
| Глава 1.5. ES.10. Объявляйте имена по одному в каждом объявлении | 73 |
| Позвольте представить..... | 73 |
| Обратная совместимость | 76 |
| Пишите более ясные объявления..... | 77 |
| Структурное связывание | 78 |
| Подведем итог | 79 |

| | |
|---|----|
| Глава 1.6. NR.2. Функции не обязательно должны иметь только один оператор возврата..... | 80 |
| Правила меняются | 80 |
| Гарантия очистки | 83 |
| Идиома RAII | 85 |
| Пишите хорошие функции | 88 |
| Подведем итог | 90 |

ЧАСТЬ II НЕ НАВРЕДИТЕ СЕБЕ

| | |
|---|-----|
| Глава 2.1. P.11. Инкапсулируйте беспорядочные конструкции, а не разбрасывайте их по всему коду..... | 92 |
| Все одним глотком | 92 |
| Что означает инкапсулировать запутанную конструкцию | 94 |
| Назначение языка и природа абстракции..... | 96 |
| Уровни абстракции..... | 100 |
| Абстракция путем рефакторинга и проведения линии | 101 |
| Подведем итог | 102 |
| Глава 2.2. I.23. Минимизируйте число параметров в функциях | 103 |
| Сколько они должны получать? | 103 |
| Упрощение через абстрагирование | 105 |
| Делайте так мало, как возможно, но не меньше | 107 |
| Примеры из реальной жизни | 109 |
| Подведем итог | 111 |
| Глава 2.3. I.26. Если нужен кросс-компилируемый ABI, используйте подмножество в стиле C | 112 |
| Создавайте библиотеки..... | 112 |
| Что такое ABI..... | 114 |
| Сокращайте до абсолютного минимума..... | 115 |
| Распространение исключений..... | 118 |
| Подведем итог | 119 |
| Глава 2.4. C.47. Определяйте и инициализируйте переменные-члены в порядке их объявления..... | 121 |
| Подведем итог | 131 |

| | |
|---|-----|
| Глава 2.5. СР3. Сведите к минимуму явное совместное использование записываемых данных | 132 |
| Традиционная модель выполнения..... | 132 |
| Подождите, это еще не все..... | 134 |
| Предотвращение взаимоблокировок и гонок за данными..... | 137 |
| Отказ от блокировок и мьютексов | 140 |
| Подведем итог | 143 |
| Глава 2.6. Т.120. Используйте метапрограммирование шаблонов, только когда это действительно необходимо | 144 |
| std::enable_if ==> requires..... | 152 |
| Подведем итог | 156 |

ЧАСТЬ III ПРЕКРАТИТЕ ЭТО ИСПОЛЬЗОВАТЬ

| | |
|---|-----|
| Глава 3.1. I.11. Никогда не передавайте владение через простой указатель (T*) или ссылку (T&)..... | 158 |
| Использование области свободной памяти | 158 |
| Производительность интеллектуальных указателей..... | 161 |
| Использование простой семантики ссылок | 163 |
| gsl::owner | 164 |
| Подведем итог | 167 |
| Глава 3.2. I.3. Избегайте синглтонов..... | 168 |
| Глобальные объекты — это плохо..... | 168 |
| Шаблон проектирования «Синглтон» | 169 |
| Фиаско порядка статической инициализации | 170 |
| Как скрыть синглтон..... | 173 |
| Только один из них должен существовать в каждый момент работы кода | 174 |
| Подождите минутку..... | 176 |
| Подведем итог | 179 |
| Глава 3.3. C.90. Полагайтесь на конструкторы и операторы присваивания вместо memset и memcpy | 180 |
| В погоне за максимальной производительностью..... | 180 |
| Ужасные накладные расходы конструкторов | 181 |
| Самый простой класс..... | 183 |

| | |
|--|-----|
| О чём говорит стандарт..... | 185 |
| А как же <i>constexpr</i> ? | 188 |
| Никогда не позволяйте себе недооценивать компилятор..... | 189 |
| Подведем итог | 191 |
| Глава 3.4. ES.50. Не приводите переменные с квалификатором <i>const</i> к неконстантному типу | 192 |
| Работа с большим количеством данных..... | 193 |
| Брандмауэр <i>const</i> | 195 |
| Реализация двойного интерфейса | 196 |
| Кэширование и отложенные вычисления | 198 |
| Два вида <i>const</i> | 199 |
| Сюрпризы <i>const</i> | 201 |
| Подведем итог | 202 |
| Глава 3.5. E.28. При обработке ошибок избегайте глобальных состояний (например, <i>errno</i>)..... | 204 |
| Обрабатывать ошибки сложно | 204 |
| Язык C и <i>errno</i> | 204 |
| Коды возврата | 206 |
| Исключения | 207 |
| <i><system_error></i> | 208 |
| <i>Boost.Outcome</i> | 209 |
| Почему обрабатывать ошибки так сложно | 210 |
| Свет в конце туннеля | 212 |
| Подведем итог | 214 |
| Глава 3.6. SF.7. Не используйте <i>using namespace</i> в глобальной области видимости в заголовочном файле | 215 |
| Не делайте этого | 215 |
| Неоднозначность | 216 |
| Использование <i>using</i> | 217 |
| Куда попадают символы | 219 |
| Еще более коварная проблема | 222 |
| Решение проблемы операторов разрешения области видимости | 223 |
| Изменение и расплата | 225 |
| Подведем итог | 226 |

ЧАСТЬ IV
ИСПОЛЬЗУЙТЕ НОВУЮ ОСОБЕННОСТЬ ПРАВИЛЬНО

| | |
|---|-----|
| Глава 4.1. F.21. Для возврата нескольких выходных значений используйте структуры или кортежи..... | 228 |
| Форма сигнатуры функции | 228 |
| Документирование и аннотирование | 230 |
| Теперь можно вернуть объект | 231 |
| Можно также вернуть кортеж | 234 |
| Передача и возврат по неконстантной ссылке | 237 |
| Подведем итог | 240 |
| Глава 4.2. Enum.3. Страйтесь использовать классы-перечисления вместо простых перечислений..... | 241 |
| Константы..... | 241 |
| Перечисления с заданной областью видимости..... | 244 |
| Базовый тип | 246 |
| Неявное преобразование | 247 |
| Подведем итог | 249 |
| Глава 4.3. ES.5. Минимизируйте области видимости | 250 |
| Природа области видимости | 250 |
| Область видимости блока | 251 |
| Область видимости пространства имен..... | 253 |
| Область видимости класса..... | 256 |
| Область видимости параметров функции | 258 |
| Область видимости перечисления | 259 |
| Область действия параметра шаблона..... | 260 |
| Область видимости как контекст | 261 |
| Подведем итог | 262 |
| Глава 4.4. Con.5. Используйте constexpr для определения значений, которые можно вычислить на этапе компиляции | 263 |
| От const к constexpr | 263 |
| C++ по умолчанию | 265 |
| Использование constexpr | 267 |
| inline..... | 271 |
| consteval..... | 272 |

| | |
|---|-----|
| constinit | 273 |
| Подведем итог | 275 |
| Глава 4.5. Т.1. Используйте шаблоны для повышения уровня абстрактности кода..... | 276 |
| Повышение уровня абстракции | 278 |
| Шаблоны функций и абстракция..... | 280 |
| Шаблоны классов и абстракция | 283 |
| Выбор имени — сложная задача | 285 |
| Подведем итог | 286 |
| Глава 4.6. Т.10. Задавайте концепции для всех аргументов шаблона | 287 |
| Как мы здесь оказались? | 287 |
| Ограничение параметров | 290 |
| Как абстрагировать свои концепции | 293 |
| Разложение на составляющие через концепции | 296 |
| Подведем итог | 297 |
| ЧАСТЬ V | |
| ПИШИТЕ ХОРОШИЙ КОД ПО УМОЛЧАНИЮ | |
| Глава 5.1. Р.4. В идеале программа должна быть статически типобезопасной..... | 300 |
| Безопасность типов — это средство защиты в C++ | 300 |
| Объединения..... | 302 |
| Приведение..... | 304 |
| Целые без знака | 307 |
| Буферы и размеры | 310 |
| Подведем итог | 311 |
| Глава 5.2. Р.10. Неизменяемые данные предпочтительнее изменяемых | 312 |
| Неправильные значения по умолчанию | 312 |
| const в объявлениях функций | 315 |
| Подведем итог | 319 |
| Глава 5.3. I.30. Инкапсулируйте нарушения правил..... | 320 |
| Скрытие неприглядных вещей | 320 |
| Поддержание видимости, что все в порядке | 322 |
| Подведем итог | 327 |

| | |
|---|-----|
| Глава 5.4. ES.22. Не объявляйте переменные, пока не получите значения для их инициализации | 329 |
| Важность выражений и операторов | 329 |
| Объявление в стиле C | 330 |
| Объявление с последующей инициализацией | 332 |
| Максимальное откладывание объявления | 333 |
| Локализация контекстно зависимой функциональности | 335 |
| Устранение состояния | 337 |
| Подведем итог | 339 |
| Глава 5.5. Per.7. При проектировании учитывайте возможность последующей оптимизации | 340 |
| Максимальная частота кадров | 340 |
| Работа вдалеке от железа | 342 |
| Оптимизация через абстракцию | 346 |
| Подведем итог | 349 |
| Глава 5.6. E.6. Используйте идиому RAII для предотвращения утечек памяти | 350 |
| Детерминированное уничтожение | 350 |
| Утечка файлов | 353 |
| Почему это так важно | 356 |
| Все это выглядит чересчур сложным: будущие возможности | 358 |
| Где все это получить | 361 |
| Заключение | 364 |
| Послесловие | 366 |

*Брину,
Шинейд,
Рори и Лоис.*

C.47: с любовью, Гай Дэвидсон

*Джиму Эллисону, хотя он едва ли прочитает эти строки.
Весь в исследовательской работе. Хлое и Аише, которые
прежде не упоминались на первых страницах книг.*

Кейт Грегори

Избранные рекомендации по C++

P.2. Придерживайтесь стандарта ISO C++ (глава 1.1).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-Cplusplus>

P.4. В идеале программа должна быть статически типобезопасной (глава 5.1).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-typesafe>

P.10. Неизменяемые данные предпочтительнее изменяемых (глава 5.2).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-mutable>

P.11. Инкапсулируйте беспорядочные конструкции, а не разбрасывайте их по всему коду (глава 2.1).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-library>

I.3. Избегайте синглтонов (глава 3.2).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-singleton>

I.11. Никогда не передавайте владение через простой указатель (T^*) или ссылку ($T\&$) (глава 3.1).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-raw>

I.23. Минимизируйте число параметров в функциях (глава 2.2).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-nargs>

I.26. Если нужен кросс-компилируемый ABI, используйте подмножество в стиле C (глава 2.3).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-abi>

I.30. Инкапсулируйте нарушения правил (глава 5.3).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-encapsulate>

F.21. Для возврата нескольких выходных значений используйте структуры или кортежи (глава 4.1).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-out-multi>

F.51. Если есть выбор, используйте аргументы по умолчанию вместо переопределки (глава 1.2).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-default-args>

C.45. Не определяйте конструктор по умолчанию, который просто инициализирует переменные-члены; для этой цели лучше использовать внутрикласовые инициализаторы членов (глава 1.3).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-default>

C.47. Определяйте и инициализируйте переменные-члены в порядке их объявления (глава 2.4).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-order>

C.90. Полагайтесь на конструкторы и операторы присваивания вместо memset и memset (глава 3.3).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-memset>

C.131. Избегайте тривиальных геттеров и сеттеров (глава 1.4).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c131-avoid-trivial-getters-and-setters>

Enum.3. Страйтесь использовать классы-перечисления вместо простых перечислений (глава 4.2).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-class>

ES.5. Минимизируйте области видимости (глава 4.3).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-scope>

ES.10. Объявляйте имена по одному в каждом объявлении (глава 1.5).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-name-one>

ES.22. Не объявляйте переменные, пока не получите значения для их инициализации (глава 5.4).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-init>

ES.50. Не приводите переменные с квалификатором const к неконстантному типу (глава 3.4).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const>

Per.7. При проектировании учитывайте возможность последующей оптимизации (глава 5.5).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rper-efficiency>

CP.3. Сведите к минимуму явное совместное использование записываемых данных (глава 2.5).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-data>

E.6. Используйте идиому RAII для предотвращения утечек памяти (глава 5.6).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-raii>

E.28. При обработке ошибок избегайте глобальных состояний (например, errno) (глава 3.5).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-no-throw>

Con.5. Используйте constexpr для определения значений, которые можно вычислить на этапе компиляции (глава 4.4).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconst-constexpr>

T.1. Используйте шаблоны для повышения уровня абстрактности кода (глава 4.5).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-raise>

T.10. Задавайте концепции для всех аргументов шаблона (глава 4.6).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-concepts>

T.120. Используйте метапрограммирование шаблонов, только когда это действительно необходимо (глава 2.6).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-metameta>

SF.7. Не используйте using namespace в глобальной области видимости в заголовочном файле (глава 3.6).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rs-using-directive>

NR.2. Функции не обязательно должны иметь только один оператор возврата (глава 1.6).

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rnr-single-return>

Предисловие

Я получил истинное удовольствие, прочитав книгу «Красивый C++». Особенно мне понравилось, что она представляет основные рекомендации C++ совсем не так, как *C++ Core Guidelines*¹. В Руководстве правила описываются довольно кратко, густо сдобрены техническими терминами и предполагают широкое использование средств статического анализа. Эта же книга рассказывает истории, в большинстве своем взятые из игровой индустрии и основанные на эволюции кода и методов на протяжении десятилетий. В ней правила представлены с точки зрения разработчика с акцентом на преимуществах, которые можно получить, следуя этим правилам, и на неприятностях, которые могут возникнуть в результате их игнорирования. Каждое правило сопровождается более обширным обсуждением, чем может предложить C++ Core Guidelines.

Руководство стремится максимально полно охватить все правила. Естественно, подбор правил написания хорошего кода в этой книге по умолчанию не может быть полным руководством по языку. Но если ограничиться степенью полноты, необходимой для понимания обсуждаемой проблемы, то станет очевидно: Руководство C++ Core не предназначено для систематического чтения. Я рекомендую прочитать в нем разделы In: Introduction и P: Philosophy, чтобы получить представление о цели этого руководства и его концептуальной основе. А для выборочного знакомства с основными правилами создания хорошего кода, исходя из вкуса, своего видения и опыта, я советую прочитать книгу, которую вы держите в руках. Для истинных профессионалов это будет легким и увлекательным чтением. А большинство остальных разработчиков смогут узнать из нее что-то новое и полезное.

*Бъерн Страуструп (Bjarne Stroustrup),
июнь 2021 года*

¹ Руководство на английском языке свободно доступно по адресу <https://github.com/isocpp/CppCoreGuidelines>. — Примеч. пер.

Вступление

Сложность разработки программ на C++ уменьшается с выходом каждого нового стандарта и каждой новой книги. Конференций, блогов и книг более чем достаточно, и это хорошо. Но в мире не хватает инженеров с достаточно высоким уровнем подготовки для решения вполне реальных задач.

Несмотря на постоянное упрощение языка, еще многое предстоит узнать о том, как писать хороший код на C++. Бьерн Страуструп, изобретатель языка C++, и Герб Саттер (Herb Sutter), руководитель органа по стандартизации C++, посвятили много сил и времени созданию учебных материалов как для изучения C++, так и для повышения квалификации разработчиков на C++. Среди них можно назвать *The C++ Programming Language*¹ и *A Tour of C++*², а также *Exceptional C++*³ и *C++ Coding Standards*⁴.

Проблема книг даже такого скромного объема заключается в том, что они отражают состояние дел на момент их издания, тогда как C++ — это постоянно развивающийся язык. То, что было хорошим советом в 1998 году, может потерять актуальность. Развивающемуся языку нужен развивающийся путеводитель.

В руководстве содержатся простые и практические советы по улучшению стиля кода на C++, чтобы вы могли писать правильный, производительный и эффективный код с первой попытки.

¹ Stroustrup B. The C++ Programming Language, Fourth Edition. — Boston: Addison-Wesley, 2013 (*Страуструп Б. Язык программирования C++*. 4-е изд.).

² Stroustrup B. A Tour of C++, Second Edition. — Boston: Addison-Wesley, 2018 (*Страуструп Б. Язык программирования C++*. Краткий курс. 2-е изд.).

³ Sutter H. Exceptional C++. — Reading, MA: Addison-Wesley, 1999 (*Саттер Г. Новые сложные задачи на C++*).

⁴ Sutter H., Alexandrescu A. C++ Coding Standards. — Boston: Addison-Wesley, 2004 (*Саттер Г., Александреску А. Стандарты программирования на C++*).

На конференции CppCon в 2015 году Бьерном Страуструпом и Гербом Саттером в ходе их двух¹ основных докладов² был запущен онлайн-ресурс C++ Core Guidelines³. В нем содержатся простые и практические советы по улучшению стиля кода на C++, чтобы вы могли писать правильный, производительный и эффективный код с первой попытки. Это постоянно развивающийся документ, в котором нуждаются специалисты, пишущие на C++, и авторы будут рады, если вы пришлете им свои предложения с исправлениями и улучшениями. Желательно, чтобы все, от новичков до ветеранов, следовали советам из этого Руководства.

В конце февраля 2020 года на #include discord⁴ Кейт Грегори (Kate Gregory) заявила, что хотела бы издать книгу о Core Guidelines, и я, Гай Дэвидсон, сразу ухватился за эту идею. Кейт выступила на CppCon 2017⁵, где рассмотрела только десять основных правил. Я разделяю ее энтузиазм по продвижению передовых приемов программирования.

Я возглавляю отдел инженерно-технических методов в *Creative Assembly*, старейшей и крупнейшей британской студии, занимающейся разработкой игр. В нем я проработал большую часть из последних 20 с лишним лет, помогая превращать наших замечательных инженеров в великих специалистов. По нашему наблюдению, несмотря на доступность и простоту Core Guidelines, разработчики мало знакомы с этим Руководством. Мы в меру своих сил и возможностей пропагандируем его и поэтому решили написать книгу, потому что литературы о рекомендациях и правилах, перечисленных в Руководстве, пока недостаточно.

Собственно Core Guidelines можно найти по адресу <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>. Оно наполнено замечательными советами, из-за порой трудно понять, с чего начать. Можно, конечно, читать

¹ Youtube.com. 2021. CppCon 2015: *Stroustrup B. Writing Good C++14*. Доступно по адресу <https://www.youtube.com/watch?v=1OEu9C51K2A>.

² Youtube.com. 2021. CppCon 2015: *Sutter H. Writing Good C++14... By Default*. Доступно по адресу <https://www.youtube.com/watch?v=hEx5DNLWGgA>.

³ Isocpp.github.io. 2021. C++ Core Guidelines. Copyright © Standard C++ Foundation and its contributors. Доступно по адресу <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>.

⁴ #include <C++>. 2021. #include <C++>. Доступно по адресу <https://www.includecpp.org/>.

⁵ Youtube.com. 2021. CppCon 2017: *Gregory K. 10 Core Guidelines You Need to Start Using Now*. Доступно по адресу <https://www.youtube.com/watch?v=XkDEzfpdcSg>.

все подряд, сверху вниз, но понять и усвоить весь набор рекомендаций без повторного чтения практически невозможно. Рекомендации организованы в 22 основных раздела с такими названиями, как Interfaces («Интерфейсы»), Functions («Функции»), Concurrency («Конкуренция») и т. д. Каждый раздел включает отдельный набор правил и рекомендаций, иногда исчисляемых единицами, иногда десятками. Рекомендации идентифицируются по первой букве из названия раздела и номеру рекомендации в разделе. Например, «F3. Функции должны быть короткими и простыми» — третья рекомендация в разделе F, Functions.

Все рекомендации построены по одному формату. Они начинаются с названия, которое представлено как посыл к действию (делайте это, не делайте этого, избегайте этого, предпочтите это), затем следует причина и несколько примеров, а также исключения из правила (если они есть). В конце дается примечание, описывающее, как обеспечить соблюдение данного правила. Примечания по применению включают и советы авторам инструментов статического анализа, и советы разработчикам, как проводить проверку кода. Чтобы читать эти рекомендации, нужен определенный навык; решение о том, какие из них взять на вооружение, является вопросом личных предпочтений. Позвольте нам продемонстрировать, как начать пользоваться мудростью этого Руководства.

В C++ есть свои острые углы, есть и свои пыльные тайники, которые не так часто посещаются в современном C++. Мы хотим увести вас от них и показать, что C++ совсем не трудный и не сложный язык и его использование вполне можно доверить большинству разработчиков.

О КНИГЕ

В этой книге мы предлагаем 30 избранных рекомендаций по программированию на C++, которые сами считаем лучшими. Мы подробно объясняем эти рекомендации в надежде, что вы будете придерживаться хотя бы их, если решите не исследовать остальные советы в Core Guidelines. Рекомендации, которые мы отобрали, не обязательно являются самыми важными, но они, вне всяких сомнений, изменят ваш код к лучшему. Конечно, полезно ознакомиться также со множеством других рекомендаций и следовать им. Мы надеемся, что вы прочтете остальные советы и попробуете их

применить к своему коду. Руководство предназначено для всех разработчиков на C++, с любым уровнем опыта, и эта книга адресована тому же кругу людей. Материал не усложняется по ходу книги, и главы можно читать в любом порядке. Они не зависят друг от друга, хотя кое-где присутствуют ссылки на другие главы. Объем каждой главы примерно 3000 слов, так что вы можете считать эту книгу скорее томиком для чтения на диване по вечерам, чем учебником. Цель книги не в том, чтобы научить вас программировать на C++, а чтобы познакомить с советами, как улучшить свой стиль.

Мы разделили рекомендации на пять частей по шесть глав в соответствии с первоначальной презентацией Кейт Грегори на CppCon в 2017 году.

В части I «Bikeshedding — это плохо» мы представляем рекомендации, которые помогут выбрать верное решение из нескольких вариантов и двигаться дальше с минимумом суеты и споров. Понятие bikeshedding¹ (bike — «велосипед», shed — «навес», то есть bikeshedding — «ставить велосипед под навес». — Примеч. ред.) заимствовано из «закона тривиальности» К. Норткота Паркинсона (C. Northcote Parkinson), согласно которому члены организации нередко придают несоразмерное значение тривиальным вопросам, таким как выбор цвета навеса для велосипедов, вместо критериев испытаний атомной электростанции, к которой он, навес, прилагается. Какова причина тривиального подхода? Да просто велосипед — это единственная вещь, о которой все хоть что-то знают.

В части II «Не навредите себе» мы предоставляем рекомендации по предотвращению травм при написании кода. Одна из проблем, связанных с остаточной сложностью C++, заключается в возможности в некоторых ситуациях выстрелить себе в ногу. Например, несмотря на допустимость заполнять список инициализации конструктора в любом порядке, никогда не следует этого делать.

Часть III называется «Прекратите это использовать» и касается элементов языка, сохраненных ради обратной совместимости, а также советов, которые раньше считались цennыми, но утеряли свою значимость благодаря нововведениям в языке. По мере развития C++ то, что раньше казалось хорошей

¹ 2021. Доступно по адресу <https://exceptionnotfound.net/bikeshedding-the-daily-software-anti-pattern/>.

идеей, сейчас может оказаться не таким ценным, как предполагалось изначально. Процесс стандартизации исправляет эти моменты, но вы должны знать о них, потому что можете столкнуться с ними, сопровождая старые проекты. Язык C++ гарантирует обратную совместимость: код, написанный 50 лет назад на C, будет компилироваться и сегодня.

Цель книги не в том, чтобы научить вас программировать на C++, а чтобы познакомить с советами, как улучшить свой стиль.

Далее следует часть IV под заголовком «Используйте новую особенность правильно». Такие особенности, как концепции, `constexpr`, структурное связывание и т. д., требуют осторожности при развертывании. Опять же C++ — это развивающийся стандарт, и с каждым выпуском появляется что-то новое, требующее освоения для поддержки. Хотя эта книга не ставит своей целью научить вас новым возможностям, которые определяются стандартом C++20, эти рекомендации помогут вам понять, как воспринимать их.

Часть V, последняя, называется «Пишите хороший код по умолчанию». Здесь описываются простые рекомендации, которые, если следовать им, помогут вам писать хороший код, не сильно задумываясь о происходящем. Придерживаясь их, вы научитесь писать красивый идиоматический код на C++, и он будет понят и оценен вашими коллегами.

На протяжении всей книги, как и в любом хорошем повествовании, возникает и развивается обсуждение разных тем. Мы, авторы, испытывали особое удовольствие от возможности увидеть мотивы, стоящие за рекомендациями, и проанализировать широкое их применение. Надеемся, что такое же удовольствие испытаете и вы при чтении. Многие рекомендации, если внимательно присмотреться, просто иначе формулируют некоторые из фундаментальных истин разработки программного обеспечения. Знание этих истин значительно улучшит ваши навыки программирования.

Мы искренне надеемся, что эта книга вам понравится и принесет пользу¹.

¹ Как уже понятно из введения и вступительного слова авторов, эта книга не просто о программировании, а о программировании, поданном через призму другой книги, название которой — C++ Core Guideline. Чтобы отличать ее от прочих руководств по языку C, упоминаемых далее по тексту, она фигурирует как Руководство — с прописной буквы. Все остальные пособия и труды называются в книге просто руководствами. — Примеч. ред.

КОД ПРИМЕРОВ

Все примеры кода доступны на сайте Compiler Explorer. Мэтт Годболт (Matt Godbolt) любезно зарезервировал постоянные ссылки для каждой главы, которые формируются путем присоединения номера главы к базовому адресу <https://godbolt.org/z/cg30-ch>. Например, ссылка <https://godbolt.org/z/cg30-ch1.3> приведет вас к примерам кода для главы 1.3. Мы рекомендуем начать с <https://godbolt.org/z/cg30-ch0.0>, где приводятся инструкции по использованию веб-сайта и взаимодействию с кодом.

*Гай Дэвидсон (Guy Davidson),
@hatcat01 hatcat.com.*

*Кейт Грэгори (Kate Gregory),
@gregcons gregcons.com.*

Октябрь 2021 года

Благодарности

Годы 2020-й и 2021-й были для нас довольно неспокойными, и мы хотели бы поблагодарить многих людей, так или иначе оказавших нам поддержку, когда мы работали над этой книгой.

Конечно же, мы хотим поблагодарить Бьерна Страуструпа и Герба Саттера за создание рекомендаций Core Guidelines и за то, что побудили нас написать о них. Мы также хотим поблагодарить участников CppCon за их помошь в обсуждении некоторых из этих рекомендаций.

Наши семьи оказали жизненно необходимую поддержку во время процесса написания, требующего уединения и сосредоточенности. Без поддержки со стороны самых близких нам было бы значительно тяжелее работать.

Легион друзей в дискорд-группе #include со штаб-квартирой на includecpr.org продолжает поддерживать нас в нашей повседневной практике использования C++ с июля 2017 года¹. Мы пожертвуем вам одну десятую нашего дохода от этой книги. Низкий вам поклон.

Свою помошь нам оказали также несколько членов комитета ISO WG21 C++, поддерживающего стандарт. Мы хотели бы поблагодарить Майкла Вонга (Michael Wong) и Тони ван Эрда (Tony van Eerd) за их участие.

Все примеры кода доступны на Compiler Explorer² по постоянным и понятным ссылкам, благодаря любезности Мэтта Годболта, создателя этого прекрасного сервиса. Мы выражаем ему нашу благодарность и уверяем, что его усилия оказались, несомненно, очень важными для сообщества C++.

Cppreference.com³ послужил нам отличным исследовательским инструментом при первоначальной подготовке каждой главы, поэтому мы весьма признательны создателю и владельцу сайта Нейту Колю (Nate Kohl), администраторам Повиласу Канапицкасу (Povilas Kanapickas) и Сергею

¹ <https://twitter.com/hatcat01/status/885973064600760320>

² <https://godbolt.org/z/cg30-ch0.0>

³ <https://ru.cppreference.com/w>

Зубкову (Sergey Zubkov), а также Тиму Сонгу (Tim Song) и всем другим участникам проекта и благодарим их за неустанную поддержку этого прекрасного ресурса. Они — герои сообщества.

После написания главы 3.6 нам стало ясно, что своим вдохновением мы во многом обязаны статье Артура О’Дуайера (Arthur O’Dwyer). Большое ему спасибо за его неизменную преданную службу обществу. В его блоге также можно найти рассказы о предпринимаемых им усилиях по раскрытию некоторых самых ранних приключенческих текстовых игровых программ 1970-х и 1980-х годов¹.

Для такой книги, как эта, нужна армия редакторов, поэтому мы выражаем благодарность Бьерну Страуструпу, Роджеру Орру (Roger Orr), Клэр Макрей (Clare Macrae), Артуру О’Дуайеру, Ивану Чукичу (Ivan Čukić), Райнери Гrimmu (Rainer Grimm) и Мэтту Годболту.

Неоценимую помощь нам оказала и команда Addison-Wesley, поэтому мы выражаем огромную благодарность Грегори Доенчу (Gregory Doench), Одри Дойл (Audrey Doyle), Асвани Кумару (Aswini Kumar), Менке Мехте (Menka Mehta), Джули Нахил (Julie Nahil) и Марку Таберу (Mark Taber).

¹ <https://quuxplusone.github.io/blog>

Об авторах

Дж. Гай Дэвидсон впервые познакомился с компьютерами благодаря Acorn Atom в 1980 году. Еще будучи подростком, он писал игры для различных домашних компьютеров: Sinclair Research ZX81 и ZX Spectrum, а также Atari ST. После получения степени по математике в Университете Сассекса он увлекся театром и играл на клавишных инструментах в соул-группе. В начале 1990-х стал заниматься разработкой приложений для презентаций, а в 1997-м перешел в игровую индустрию, начав работать в Codemasters в их лондонском офисе.

В 1999 году перешел в Creative Assembly, где сейчас возглавляет отдел инженерно-технических методов. Работает над франшизой *Total War*, курируя дискографию, а также формулируя и развивая стандарты программирования в команде инженеров. Входит в состав консультативных советов IGGI, группы BSI C++ и комитета ISO C++. Занимает пост ответственного за стандарты в комитете ACCU и входит в программный комитет конференции ACCU. Является модератором на дисCORD-сервере #include <C++>. Отвечает за внутреннюю политику и нормы в нескольких организациях. Его можно увидеть на конференциях и встречах по C++, особенно на посвященных добавлению методов линейной алгебры в стандартную библиотеку.

В свободное время он оказывает наставническую поддержку по вопросам программирования на C++ через Proscola и BAME in Games; помогает школам, колледжам и университетам через UKIE, STEMNet и в качестве Video Game Ambassador; практикует и преподает тай-чи в стиле У; изучает игру на фортепиано; поет первый бас в Брайтонском фестивальном хоре; управляет местным киноклубом; является членом BAFTA с правом голоса; дважды баллотировался (безуспешно) на выборах в местный совет от имени партии зеленых Англии и Уэльса; пытается выучить испанский. Иногда его можно встретить за карточным столом играющим в бридж по пенни за очко. Вероятно, у него есть и другие увлечения: он большой непоседа.

Кейт Грэгори познакомилась с программированием в Университете Ватерлоо в 1977 году и никогда не оглядалась назад с сомнением или

сожалением. Имеет степень в области химического машиностроения, что лишний раз подтверждает, что диплом не всегда говорит о наклонностях человека. На цокольном этаже ее сельского дома в Онтарио есть небольшая комната со старыми компьютерами PET, C64, домашней системой 6502 и т. д., служащими напоминаниями о более простых временах. С 1986 года вместе с мужем руководит компанией Gregory Consulting, помогая клиентам по всему миру.

Кейт выступала с докладами на пяти континентах, любит искать заковыристые головоломки и затем делиться их решением, а также проводит много времени, добровольно участвуя в различных мероприятиях, посвященных языку C++. Самым уважаемым из них является группа `#include <C++>`, которая оказывает огромное влияние на эту отрасль, делает программирование на C++ более гостеприимным и дружелюбным. Их дискорд-сервер — теплое, уютное место для изучения C++ новичками и одновременно кают-компания для совместной работы над статьями для WG21, позволяющими взглянуть по-иному на язык, который мы все используем, или же... что-то среднее между ними двумя.

Ее отрывают от клавиатуры внуки, озера и кемпинги Онтарио, весла для каноэ и дым костра, а также соблазны аэропортов по всему миру. Гурманка, игрок в настольные игры, безотказная палочка-выручалочка, не способная ответить отказом на просьбу о помощи, она так же активна в реальной жизни, как и в интернете, но менее ярка и заметна. После того как в 2016 году пережила меланому IV стадии, она стала меньше беспокоиться о том, что думают другие и чего от нее ожидают, и больше о том, чего она хочет для своего будущего. Это дает свои результаты¹.

¹ Вся книга написана от лица обоих авторов, но иногда слово в повествовании представляется одному Гаю Дэвидсону. Отступления в тексте, в которых он щедро делится с читателями личными знаниями, опытом и жизненной мудростью, оформлены как исторические экскурсы. — *Примеч. ред.*

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.



BIKESHEDDING – ЭТО ПЛОХО

https://t.me/it_boooks

Глава 1.1 Р.2. Придерживайтесь стандарта ISO C++.

Глава 1.2 F.51. Если есть выбор, используйте аргументы по умолчанию вместо перегрузки.

Глава 1.3 С.45. Не определяйте конструктор по умолчанию, который просто инициализирует переменные-члены; для этой цели лучше использовать внутриклассовые инициализаторы членов.

Глава 1.4 С.131. Избегайте тривиальных геттеров и сеттеров.

Глава 1.5 ES.10. Объявляйте имена по одному в каждом объявлении.

Глава 1.6 NR.2. Функции не обязательно должны иметь только один оператор возврата.

ГЛАВА 1.1

P.2. Придерживайтесь стандарта ISO C++

ЧТО ТАКОЕ СТАНДАРТ ISO C++

Эта книга посвящена приемам написания хорошего кода. Поэтому первый совет — придерживайтесь стандарта ISO C++. Но что это означает на практике?

История C++

C++ первоначально не был стандартизованным языком. Это было расширение языка программирования C, изобретенное Бьерном Страуструпом¹. Оно получило название «C с классами». В то время язык C тоже не был стандартизованным: Бьерн представил свое расширение в виде препроцессора под названием Cpre. Препроцессор обеспечивал поддержку классов и производных классов с общедоступными/приватными уровнями доступа, дружественными классами, перегрузкой операторов присваивания, конструкторами и деструкторами. Кроме того, препроцессор поддерживал встраиваемые функции и аргументы функций по умолчанию, а также проверку типов аргументов функций.

В 1982 году Бьерн начал работу над новым проектом под названием C++, включающим дополнительные возможности, такие как виртуальные функции, перегрузка функций и операторов, ссылки, константы и динамическое распределение памяти. Он также создал интерфейс C++ для компиляторов C под названием Cfront. Код на C++ передавался Cfront, который

¹ Stroustrup B. 1995. A History of C++: 1979–1991, www.stroustrup.com/hopl2.pdf.

затем компилировал его в код на языке С. Он также написал книгу под названием *The C++ Programming Language* (широко известную как *TCPL*), опубликованную в 1985 году. Она послужила определяющим руководством по языку C++, благодаря ей стали появляться коммерческие компиляторы.

Даже после широкого распространения этих компиляторов Бьерн продолжал работать над C++, добавляя новые возможности в то, что стало называться C++2.0. К ним относятся множественное наследование, абстрактные базовые классы, статические и константные функции-члены, защищенный уровень доступа, а также улучшения существующих возможностей. В тот период наблюдался лавинообразный рост популярности C++. По оценкам Бьерна, количество пользователей удваивалось каждые 7,5 месяца.

Появились конференции, журналы и книги, а конкурирующие между собой реализации компилятора продемонстрировали, что необходимо нечто более точное и регламентирующее, чем *TCPL*. В 1989 году Дмитрий Ленков (Dmitry Lenkov) из HP написал предложение по стандартизации C++ в Американском национальном институте стандартов (American National Standards Institute, ANSI). В нем он указывал на необходимость тщательного и подробного определения каждой особенности языка для предотвращения неконтролируемого увеличения числа диалектов. Он также определял необходимость реализации дополнительных возможностей, таких как обработка исключений и создание стандартной библиотеки. Комитет ANSI C++, X3J16, впервые собрался в декабре 1989 года. *Аннотированное справочное руководство* (Annotated Reference Manual, ARM), написанное Маргарет Эллис (Margaret Ellis) и Бьерном и опубликованное в 1990 году, стало единым основополагающим описанием всего C++. Это руководство было создано специально, чтобы ускорить начало работ по стандартизации ANSI C++.

Конечно, это был не только американский проект, и в нем приняли участие многие представители других стран. В 1991 году был созван комитет ISO C++ WG21, и с тех пор эти два комитета проводили совместные заседания с целью выработать черновой стандарт для публичного рассмотрения через четыре года, с надеждой на появление официального стандарта двумя годами после. Однако первый стандарт, ISO/IEC 14882:1998, был опубликован только в сентябре 1998-го, почти через девять лет после упомянутого первого собрания.

Но история на этом не заканчивается. Работа над исправлением ошибок в стандарте продолжалась, и в 2003 году был опубликован стандарт C++03.

Конечно же, работы по добавлению дополнительных возможностей и тогда не прекратились, было продолжено дальнейшее развитие языка. В число новых возможностей вошли: `auto`, `constexpr`, `decltype`, семантика перемещения, `for` с диапазонами, унифицированная инициализация, лямбда-выражения, правосторонние (`rvalue`) ссылки, статические утверждения, вариативные шаблоны... Список продолжал пополняться, и график разработки растягивался. В конце концов, следующая версия стандарта вышла в 2011 году, еще до того, как все его авторы успели забыть, что C++ был когда-то растущим языком.

Учитывая, что C++03 был исправлением C++98, между первым стандартом и стандартом C++11 имел место 13-летний разрыв. Стало понятно, что такой длительный период между публикациями стандартов никому не интересен, поэтому была разработана «модель транспорта»: новый стандарт будет публиковаться каждые три года, и если какая-то особенность языка не будет готова к этому моменту, отчет о ее готовности пойдет «следующим рейсом» — пополнит версию, которая будет опубликована три года спустя. С тех пор стандарты C++14, C++17 и C++20 выходили согласно графику.

ИНКАПСУЛЯЦИЯ ВАРИАЦИЙ

Вариации в окружении времени выполнения

В стандарте очень мало говорится о требованиях к окружению, в котором выполняется программа на C++. Операционная система не регламентируется. Хранилища файлов не являются обязательными. Экран тоже необязателен. Получается, что программа, написанная для типичного окружения рабочего стола, может нуждаться во вводе с помощью мыши и выводе в оконном режиме, что потребует специального кода для каждой конкретной системы.

Написание полностью переносимого кода для такой программы невозможно. Стандарт ISO C++ имеет очень маленькую библиотеку по сравнению с такими языками, как C# и Java. Это всего лишь спецификация для разработчиков, реализующих стандарт ISO C++. Все дело в том, что стандартные библиотеки C# и Java предоставляются владельцами языка, но C++ не имеет финансируемой организации по разработке библиотек. Вы должны использовать уникальные особенности каждой целевой среды для поддержки тех частей функционала, которые недоступны в стандартной библиотеке языка. Обычно они предлагаются в виде заголовочного файла и библиотеки. Как правило, таких файлов очень много в каждой системе.

Насколько возможно, прячьте их за вашими собственными интерфейсами. Минимизируйте количество вариаций между версиями кодовой базы, предназначенными для разных систем.

Например, вам может понадобиться узнать, нажимается ли конкретная клавиша на клавиатуре. Один из возможных подходов — применение препроцессора для определения используемой платформы и вызов соответствующего фрагмента кода, например:

```
#if defined WIN32
auto a_pressed = bool{GetKeyState('A') & 0x8000 != 0};
#elif defined LINUX
auto a_pressed = /* в действительности здесь следует большой фрагмент кода */
#endif
```

Очень неуклюжее решение: оно работает на неправильном уровне абстракции. Код, относящийся к Windows и Linux¹, должен находиться в отдельных файлах, соответствующих системе, и импортироваться через заголовочные файлы, чтобы использующий их вызов выглядел так:

```
auto a_pressed = key_state('A');
```

Функция `key_state` представляет собой интерфейс, инкапсулирующий это расширение. Реализация делает все, что нужно для соответствующей платформы, и ваш поток управления не захламляется мусором в виде макросов препроцессора. Разделение каждой реализации по разным файлам обеспечивает дополнительную поддержку этой абстракции.

Вариации на уровне языка C++ и компилятора

Разработчики компиляторов C++ должны максимально полно и точно поддерживать стандарт, если намерены объявить свой компилятор соответствующим стандарту. Однако это требование не связывает им руки и оставляет открытой дверь для добавления дополнительных возможностей или расширений. Например, в GCC были добавлены дополнительные свойства типов (`type trait`), такие как `_has_trivial_constructor` и `_is_abstract`, до того как они появились в стандарте. Оба присутствовали в библиотеке свойств типов, начиная с C++11, под другими именами: `std::is_trivially_constructible` и `std::is_abstract`.

Обратите внимание, что имя `_is_abstract` начинается с двойного подчеркивания: имена, начинающиеся с двойного подчеркивания, зарезервированы

¹ <https://stackoverflow.com/questions/41600981/how-do-i-check-if-a-key-is-pressed-on-c>

стандартом для разработчиков компилятора. Разработчикам *не* разрешается добавлять новые идентификаторы в пространство имен `std`, так как впоследствии они могут быть добавлены в стандарт с совершенно другим значением. На практике это означает, что программист, работая на C++, может ненамеренно написать код, который выглядит как использующий стандартные функции, но в действительности использует функции, характерные для специфического компилятора. Хороший способ защититься от таких случайностей — написать и протестировать свою программу с несколькими компиляторами и операционными системами, чтобы обнаружить не закрепленный в стандарте код.

Две упомянутые выше функции были реализованы по причине их полезности для метапрограммирования. В действительности они оказались настолько полезными, что позже были добавлены в стандарт. Многие части стандарта, касающиеся как самого языка, так и стандартной библиотеки, появились сначала как дополнительные возможности в популярных инструментах и библиотеках. Иногда использование нестандартных функций неизбежно.

Расширения для C++

Некоторые разработчики библиотек добавляют свои расширения. Например, библиотека Qt¹ использует так называемые сигналы и слоты для организации взаимодействий между объектами. С этой целью в библиотеку добавлены три символа: `Q_SIGNALS`, `Q_SLOTS` и `Q_EMIT`. В исходном коде эти ключевые слова выглядят как любые другие ключевые слова языка. Qt предоставляет инструмент под названием `moc`, который анализирует эти ключевые слова и преобразует их в конструкции, понятные компилятору C++, точно так же, как раньше Cfront преобразовывал код на C++ в конструкции, понятные компиляторам языка C.

Следует иметь в виду, что стандарт предлагает то, чего нет в этих расширениях: строго определенную семантику. Стандарт ISO C++ однозначен, и это одна из причин, почему его так трудно читать. Можно без ограничений использовать расширения языка, но не забывать о факторе переносимости. В частности, Qt прилагает титанические усилия для достижения переносимости кода между различными платформами. Однако никто не гарантирует, что эти расширения будут присутствовать в других реализациях или что они будут иметь там такое же значение.

¹ <https://doc.qt.io/>

Защита заголовочных файлов

Рассмотрим пример с директивой `#pragma once`. Эта простая директива приказывает компилятору не подключать заголовочный файл во второй раз и тем самым сокращает время, затрачиваемое на компиляцию единицы трансляции. Все компиляторы, которые авторы использовали за последние 20 лет, реализуют эту директиву `pragma`, но задумаемся: что она означает на самом деле? Может, «остановить синтаксический анализ, пока не будет достигнут конец файла»? Или «не открывать этот файл во второй раз»? Получается, притом что видимый эффект одинаков, его значение не имеет точного определения для всех платформ без исключения.

В результате вы как разработчики не можете быть уверенными, что значение чего-либо сохранится на разных платформах. Даже если сейчас вы в безопасности, то не можете гарантировать, что останетесь в безопасности в будущем. Полагаться на такую нестандартную возможность — все равно что полагаться на ошибку. Это опасно, потому что она может быть изменена или исправлена в любое время (впрочем, смотрите Закон Хайрама¹). В этом случае Руководство рекомендует вместо `#pragma once` использовать средства защиты заголовочных файлов, как описано в «SF.8. Защищайте от повторного подключения все заголовочные файлы». При наличии такой защиты мы точно знаем, что произойдет.

Вариации в основных типах

Реализации операционных систем не единственный вид системных вариаций. Как вы, возможно, знаете, размеры арифметических типов, таких как `int` и `char`, не стандартизированы. Вы можете считать, что тип `int` имеет размер 32 бита, но были времена, когда `int` имел размер 16 бит. Иногда в программах бывает нужен тип размером ровно 32 бита, поэтому, опасаясь совершить ошибку (предположив, что `int` всегда будет иметь размер 32 бита: размер этого типа уже изменился один раз, так почему бы ему не измениться снова?), разработчики вынуждены были использовать заголовки реализации, чтобы выяснить, какой тип имеет нужный размер, и определить псевдоним для этого типа:

```
typedef __int i32; // это древний способ добиться желаемого:  
                   // не используйте его
```

¹ Доступно по адресу <https://www.hyrumsLaw.com/>.

Здесь представлен идентификатор `i32`, который играл роль псевдонима для типа с именем `_int`, определяемого платформой. Такой шаг придал этому коду некоторую безопасность: если бы проект пришлось переносить на другую платформу, то можно было бы узнать название 32-битного целочисленного типа со знаком, определяемого этой платформой, и просто обновить определение `typedef`.

Когда был выпущен следующий стандарт, в данном случае C++11, в библиотеку были добавлены новые типы в заголовочном файле `<cstdint>`, которые определяли целочисленные типы фиксированного размера. В этой связи появилась возможность обновить определение и получить двойную выгоду:

```
using i32 = std::int32_t;
```

Во-первых, новый тип можно использовать для проверки своего определения в будущем: тип, которому присвоен псевдоним, является частью стандарта, и крайне маловероятно, что он изменится, потому что обратная совместимость очень важна для языка и сохраняется незыблемой. Это объявление типа останется верным и в последующих версиях стандарта (действительно, прошло девять лет и вышло три стандарта, а этот код все еще компилируется).

Во-вторых, разработчики смогли перейти на использование нового ключевого слова `using`, которое позволяет использовать стиль записи слева направо. В нем идентификатор и его определение разделены знаком равенства. Этот стиль также можно увидеть в примерах использования ключевого слова `auto`:

```
auto index = i32{0};
```

Определяемый идентификатор указывается слева от знака равенства, а его определение — справа.

Когда появились достаточно мощные инструменты рефакторинга, нашлись разработчики, которые сделали решительный шаг и заменили все экземпляры `i32` на `std::int32_t`, чтобы минимизировать неоднозначность прочтения кода.

Нормативные ограничения

Следует отметить, что иногда просто невозможно использовать стандарт ISO C++. Не из-за какого-то недостатка в библиотеке или отсутствующей особенности в языке, а из-за того, что внешнее окружение запрещает

использование определенных функций. Это может быть обусловлено нормативными причинами или несовершенством реализации платформы, для которой вы разрабатываете приложения.

Например, в некоторых отраслях запрещается динамически распределять память в функциях, производительность которых критически важна. Распределение памяти — это недетерминированное действие, которое, кроме всего прочего, может вызвать исключение из-за нехватки памяти; то есть в динамике невозможно точно спрогнозировать, сколько времени займет выполнение динамического запроса памяти. Или, например, в некоторых областях и по тем же причинам запрещена генерация исключений, а это опять же сразу налагает запрет на динамическое распределение памяти, потому что `std::operator new` генерирует исключение `std::bad_alloc` при ошибке. Из-за подобных ситуаций ощущается необходимость расширить и адаптировать Core Guidelines к конкретной среде.

И наоборот, в некоторых областях запрещено использование библиотек, не прошедших сертификацию отраслевым регулирующим органом. Например, использование библиотеки Boost¹ может быть проблематичным в некоторых окружениях. Очевидно, что становится все более необходимо шире и чаще использовать стандарт ISO C++.

ИЗУЧЕНИЕ СТАРЫХ СПОСОБОВ

Обратная совместимость в C++

Важно помнить, откуда взялся этот язык, а также что послужило мотивом его развития. В текущем проекте одного из авторов есть код, который был им написан в 2005 году. Код выглядит немного странно для современных программистов, потому что использует давно заброшенные парадигмы, никаких объявлений `auto`, никаких лямбда-выражений: он является артефактом истории исходного кода.

Тем не менее он по-прежнему прекрасно компилируется и работает. В своей карьере вы столкнетесь с кодом разных эпох. Важно использовать последний стандарт и собирать код с помощью самого свежего компилятора, который только сможете найти, но также важно знать, откуда взялся язык и что было раньше, хотя бы для того, чтобы уметь предвидеть будущее.

¹ <https://www.boost.org>

Иногда нет возможности использовать последнюю версию стандарта. При разработке встраиваемых систем регламент, сертификация систем или устаревшая инфраструктура могут вынудить вас использовать C++11 или даже C++98. C++ полагается на обратную совместимость. Он обратно совместим с C и с предыдущими стандартами. Стабильность в течение десятилетий является преимуществом. Это одна из сильных сторон C++: миллиарды строк кода, написанных во всем мире, до сих пор компилируются с помощью современных компиляторов, пусть иногда и с небольшими корректировками. В какой-то момент вас могут попросить обеспечить поддержку такому коду.

Прямая совместимость и Y2K

С другой стороны, пишите код на века. В конце прошлого века в большинстве старых компьютерных программ, существующих в мире, была обнаружена проблема: для обозначения года использовались только две цифры¹. В те времена память была в дефиците и эффективнее было хранить 74, а не 1974. Разработчики думали: «Эта программа не будет работать через 25 лет, ее наверняка заменят». Ах, какой пессимизм, а может быть, и оптимизм, это зависит от точки зрения.

Как только дата пересекла 2000 год, номер года стал представляться как 00, что отрицательно сказалось на вычислениях временных интервалов, расчетах процентных платежей и вообще на *всем*, что распределено во времени.

Стабильность и преемственность кода в течение десятилетий являются преимуществом.

Эта ошибка известна как Y2K или ошибка тысячелетия. Наступил звездный час старых подрядчиков, которые путешествовали по компьютерам мира, дорабатывая 25-летние системы за немалые деньги. Хотя катастрофа в значительной степени была предотвращена, потому что проблема была выявлена загодя и имелось достаточно инженеров, чтобы все исправить.

Однако если бы инженеры-разработчики ПО в свое время планировали будущее, предполагали, что их код будет работать «вечно», и в то время, когда они писали код, четырехзначные целые числа занимали бы столько же места, сколько занимали двузначные числа, то этих неприятностей и тревог

¹ <https://www.britannica.com/technology/Y2K-bug>

удалось бы избежать. Было бы ясно, что двух цифр *недостаточно* для представления всех дат, которые могут потребоваться, и понадобилось бы по крайней мере три цифры, а еще лучше — четыре, просто чтобы легче пересечь рубеж тысячелетий.

Кстати, это не единственная проблема с датами. В Linux существовала аналогичная проблема с измерением времени в секундах от начала эпохи Unix 1 января 1970 года. Это число хранилось как 32-битное целое число со знаком, то есть оно стало бы отрицательным 19 января 2038 года. Мы говорим «стало бы», потому что в версии Linux 5.6 эта ситуация была решена.

Итак, важной парой профессиональных навыков является написание кода с прицелом на будущее и умение читать код из прошлого.

СЛЕДИТЕ ЗА ПОСЛЕДНИМИ ИЗМЕНЕНИЯМИ В СТАНДАРТЕ

C++ постоянно развивается. С появлением каждой новой версии стандарта в языке появляются новые возможности и дополнения к библиотекам. Простое механическое использование самых последних новинок не несет особой ценности, их следует использовать там, где они приносят определенную и конкретную пользу. Однако сообществу C++ очень повезло: у него есть много отличных учителей и толкователей, готовых открыть и объяснить все эти новинки. Инженерам нужно держать руку на пульсе тенденций и тонкостей этого развития. Поиск необходимых ресурсов можно вести в четырех направлениях.

IsoCpp

Прежде всего, это isocpp.org¹. Это дом C++ в интернете, которым управляет Standard C++ Foundation — некоммерческая организация Washington 501(c) (6), целью которой является поддержка сообщества разработчиков программного обеспечения на C++ и содействие пониманию и использованию современного стандарта C++ на всех платформах и компиляторах. На этом сайте вы найдете обзор C++, написанный Бьерном, огромный сборник вопросов и ответов по C++, подробную информацию

¹ <https://isocpp.org/about>

о том, как можно принять участие в процессе стандартизации, и регулярно обновляемый список последних материалов, опубликованных членами сообщества C++. Этот список послужит вам отличной отправной точкой для поиска других статей в блогах.

Конференции

Второй источник – ежегодные конференции, проводимые по всему миру. На этих конференциях принято записывать все выступления и публиковать их на YouTube для бесплатного просмотра. Это действительно удивительный, очень активный ресурс, и довольно сложно хотя бы просто идти в ногу с публикуемыми материалами из года в год, не отставать и оставаться в курсе всех публикаций.

Конференция CppCon находится в ведении Standard C++ Foundation. Она проходит ранней осенью в США, в городе Аврора, штат Колорадо, и генерирует около 200 часов контента. Ассоциация пользователей С и С++ (Association of C and C++ Users, ACCU) проводит свою ежегодную конференцию в Бристоле, Великобритания, каждую весну, а иногда и осенью. Она фокусируется в основном на C++, но иногда на ней обсуждаются более широкие темы, связанные с программированием. Ассоциация генерирует почти 100 часов контента. Ежегодно в ноябре в Берлине, Германия, проводится встреча пользователей C++, которая генерирует около 50 часов контента. Конечно, охватить такой объем информации у вас едва ли получится, потому что просмотр хотя бы одного доклада в день в течение, например, года займет у вас массу времени, фактически большую часть этого года. А ведь есть еще множество других небольших конференций, которые проходят в таких странах, как Австралия, Беларусь, Израиль, Италия, Польша, Россия, Испания...

Другие источники

Помимо блогов и конференций, существует также множество книг. Некоторые из них будут упоминаться в ссылках по всему тексту этой книги, как и цитаты из выступлений на конференциях.

Наконец, на разных чат-серверах, таких как Discord и Slack¹, доступны ежедневные обсуждения. Сервер Discord модерируется группой #include²

¹ <https://cpplang.slack.com>

² <https://www.includecpp.org>

и предназначен для программистов на C++. К настоящему времени там образовалось очень гостеприимное сообщество.

Имея так много доступных ресурсов, вы сможете идти в ногу с развитием стандарта C++. Разработка кода с опорой на стандарт ISO C++ доступна каждому. Это важно не только для тех, кто в будущем будет сопровождать код, кем бы они ни были, включая вас самих, но и для будущих клиентов-заказчиков вашего кода. Язык C++ широко используется во многих областях: в торговле, промышленности и общественных организациях. Применение последовательного и надежного подхода к разработке кода имеет глобальное значение. Итак, будучи ответственными профессионалами, сделайте шаг вперед, поступайте правильно и придерживайтесь стандарта ISO C++.

ГЛАВА 1.2

F.51. Если есть выбор, используйте аргументы по умолчанию вместо перегрузки

https://t.me/it_boooks

ВВЕДЕНИЕ

Проектирование API — ценный навык. Разбивая задачу на составляющие, вы должны выделить абстракции и спроектировать для них интерфейс, дав клиенту-пользователю четкие и недвусмысленные инструкции в виде совершенно очевидного набора функций с тщательно подобранными именами. Есть такая рекомендация, которая гласит, что код должен быть самодокументируемым. Несмотря на кажущуюся чрезмерную амбициозность цели, именно в проектировании API вы должны приложить все силы для ее достижения.

Базы кода постоянно растут. Это неизбежно, и от этого никуда не деться. Со временем обнаруживается и кодируется все больше абстракций, решается больше задач, и сама предметная область неуклонно расширяется, чтобы затребовать и затем вместить больше вариантов использования программ. Это совершенно正常но. Это часть типичного процесса разработки и проектирования.

По мере добавления дополнительных абстракций в базу кода свою уродливую голову поднимает проблема однозначного именования. Подбор правильных, говорящих имен — сложная задача. И вы не раз убедитесь в этом в своей карьере программиста. Иногда возникает необходимость позволить клиенту (а часто им являетесь вы сами) сделать вроде бы то же самое, но немного по-другому.

В таких случаях перегрузка может показаться хорошей идеей. Различие между двумя абстракциями может заключаться лишь в передаваемых им аргументах; во всем остальном они семантически идентичны. Перегрузка функций позволяет повторно использовать имя функции, но с другим набором параметров. Однако если они действительно семантически идентичны, то, может быть, лучше выразить эту разницу в форме аргументов по умолчанию? Если это действительно так, то ваш API станет проще для понимания.

Прежде чем начать обсуждение, мы хотим напомнить вам о разнице между параметром и аргументом: аргумент — это то, что передается в функцию. Объявление функции включает список параметров, из которых один или несколько могут быть снабжены аргументами (значениями) по умолчанию. Не существует такого понятия, как параметр по умолчанию.

ДОРАБОТКА ВАШИХ АБСТРАКЦИЙ: ДОПОЛНИТЕЛЬНЫЕ АРГУМЕНТЫ ИЛИ ПЕРЕГРУЗКА?

Рассмотрим для примера следующую функцию:

```
office make_office(float floor_space, int staff);
```

Эта функция возвращает экземпляр `office` — объект, представляющий собой офисное здание площадью `floor_space` квадратных метров и с кабинетами для `staff` сотрудников. Это одноэтажное здание с кухонными и туалетными помещениями и соответствующим количеством кофемашин, столов для настольного тенниса и массажных кабинетов. Однажды было решено расширить предметную область и добавить возможность моделирования двухэтажных офисных зданий. Это несколько усложнило ситуацию, так как двухэтажная модель предполагает определение путей эвакуации, более сложную схему кондиционирования воздуха, наличие лестниц в нужных местах и, конечно же, горки между этажами или, может быть, пожарного столба. К тому же нужно сообщить функции-конструктору, что она должна создать модель двухэтажного офисного здания. Это можно сделать с помощью третьего параметра:

```
office make_office(float floor_space, int staff, bool two_floors);
```

Проблема в том, что в этом случае придется просмотреть весь код и добавить `false` во все вызовы `make_office`. С другой стороны, можно определить

аргумент по умолчанию `false` для последнего параметра, и тогда ничего менять не придется. Вот как это выглядит:

```
office make_office(float floor_space, int staff, bool two_floors = false);
```

Одна короткая перекомпиляция — и все в порядке. К сожалению, демоны расширения предметной области еще не закончили: оказывается, одноэтажным офисным зданиям иногда требуется давать названия. Вы, как отзывчивый на вызовы и просьбы заказчика инженер, решаете расширить список параметров функции:

```
office make_office(float floor_space, int staff, bool two_floors = false,
                   std::string const& building_name = {});
```

Вы переопределяете функцию и замечаете одну неприятность. Функция принимает четыре аргумента, причем последний необходим, только если третий аргумент `false` и из-за этого функция выглядит запутанной и сложной. Вы решаете добавить перегруженную версию функции:

```
office make_office(float floor_space, int staff, bool two_floors = false);
office make_office(float floor_space, int staff,
                   std::string const& building_name);
```

Теперь у вас есть то, что известно как набор перегруженных функций. И каждый раз, встретив ссылку на имя функции, компилятор должен выбрать, какую реализацию выбрать, а для этого проверить типы переданных аргументов. Клиент вынужден вызывать правильную функцию, когда нужно идентифицировать здание. Наличие идентификации подразумевает одноэтажный офис.

Например, представьте, что в некотором клиентском коде предпринята попытка создать офис площадью 24 000 квадратных метров для 200 сотрудников. Офис расположен в одноэтажном здании с названием Eagle Heights. Вот как должен выглядеть соответствующий вызов:

```
auto eh_office = make_office(24000.f, 200, "Eagle Heights");
```

Конечно, вы должны гарантировать соблюдение определенной семантики в каждой функции и обеспечить, чтобы все функции действовали одинаково. Это тяжкое бремя сопровождения. Возможно, уместнее реализовать единственную функцию и потребовать от вызывающей стороны явно обозначать свой выбор.

Мы уже слышим, как вы говорите: «Постойте! А если написать приватную реализацию функции? В таком случае можно гарантировать единообразие

моделирования, просто вызывая приватную реализацию из перегруженных версий, и все будет в порядке».

Вы были бы правы, если бы не одно но. Клиенты могут с подозрением отнестись к двум функциям. Их может насторожить возможность несогласованности реализаций. Излишняя осторожность с их стороны может вселить в них страх и опасения. Одна функция с двумя аргументами по умолчанию для переключения между алгоритмами выглядит более надежно.

И снова мы слышим ваше возражение: «Это смешно! Я пишу качественный код, и мои клиенты доверяют мне. У меня весь код охвачен модульными тестами, и все в порядке. Вот уж спасибо так спасибо!»

К сожалению, даже если вы действительно пишете код высочайшего качества, это не обязательно относится к вашим клиентам. Взгляните еще раз на инициализацию `eh_office` и проверьте себя, сможете ли вы заметить ошибку. А другой человек сможет? Подумайте, а пока мы поговорим о разрешении перегрузки кода (как выбираются перегруженные версии).

ТОНКОСТИ РАЗРЕШЕНИЯ ПЕРЕГРУЗКИ

Разрешение перегрузки — сложная задача для освоения. Почти 2 % стандарта C++20 посвящены определению работы механизма разрешения перегрузок. Вот краткий обзор.

Когда компилятор встречает вызов функции, он должен определить, на какую из функций этот вызов ссылается. Перед этим компилятор составляет список всех идентификаторов. Возможно, что в программе имеется несколько функций с одинаковыми именами, но с разными параметрами — набор перегруженных версий. Как компилятор определяет, какую из них вызывать?

Сначала он отбирает функции с тем же количеством параметров, с меньшим количеством и с параметром-многоточием или с большим количеством параметров, среди которых избыточные параметры имеют аргументы по умолчанию. Если какой-либо из кандидатов имеет предложение `requires` (`requires clause`, нововведение, появившееся в C++20), то оно должно быть удовлетворено. Ни один правосторонний (`rvalue`) аргумент не должен соответствовать неконстантному левостороннему (`lvalue`) параметру, и любой левосторонний (`lvalue`) аргумент не должен соответствовать ссылочному правостороннему (`rvalue`) параметру. Каждый аргумент должен иметь возможность быть преобразованным в соответствующий параметр посредством неявной последовательности преобразований.

В нашем примере компилятору передаются две версии `make_office`, отличающиеся третьим параметром. Одна принимает логическое значение, которое по умолчанию равно `false`, а вторая — `std::string const&`. По количеству параметров обе версии соответствуют операции инициализации `eh_office`.

Ни в одной из этих функций нет предложения `requires`, поэтому можно пропустить этот шаг. Точно так же нет ничего экзотического в привязках ссылок.

Наконец, каждый аргумент должен быть преобразован в соответствующий параметр. Первые два аргумента не требуют преобразования. Третий аргумент — это `char const*`, который, очевидно, преобразуется в `std::string` через неявный конструктор, являющийся частью интерфейса `std::string`. Но, к сожалению, это еще не все.

Когда имеется несколько перегруженных версий функции, они ранжируются по параметрам, чтобы упростить поиск наиболее подходящей. Версия F1 считается предпочтительнее версии F2, если неявные преобразования для всех аргументов F1 не хуже, чем у F2. Кроме того, в F1 должен быть хотя бы один параметр, неявное преобразование которого лучше соответствующего неявного преобразования в F2.

Слово «лучше» настораживает. Как ранжируются последовательности неявных преобразований?

Существует три типа последовательностей неявных преобразований: стандартная, определяемая пользователем и последовательность преобразований с многоточием.

Стандартная последовательность имеет три ранга: точное соответствие, продвижение и преобразование. Точное соответствие означает отсутствие необходимости преобразования и является предпочтительным рангом. Это также может означать преобразование левостороннего (`lvalue`) аргумента в правосторонний (`rvalue`).

Продвижение означает расширение представления типа. Например, объект типа `short` может быть продвинут до объекта типа `int` (такое преобразование называется целочисленным продвижением), а объект типа `float` может быть продвинут до объекта типа `double`, что известно как продвижение с плавающей точкой.

Преобразования отличаются от продвижения возможностью изменения значения, что может отрицательно сказаться на точности. Например,

значение с плавающей точкой можно преобразовать в целое число, округлив до ближайшего целого. Кроме того, целочисленные значения и значения с плавающей точкой, перечисления без указания области видимости, указатели и типы указателей на члены могут быть преобразованы в логическое значение. Эти три ранга являются концепциями, унаследованными от языка C, и от них невозможно отказаться из-за необходимости поддерживать совместимость с C.

Это частично охватывает стандартные последовательности преобразований. Преобразования, определяемые пользователем, выполняются двумя способами: либо с помощью неявного конструктора, либо с помощью неявного оператора преобразования. Именно этот тип преобразований мы ожидаем в нашем примере: наш аргумент `char const*` преобразуется в `std::string` через неявный конструктор, который принимает `char const*`. Это так же очевидно, как нос на вашем лице. Но с какой целью мы втянули вас в это обсуждение особенностей разрешения перегрузок?

ВЕРНЕМСЯ К ПРИМЕРУ

В примере выше клиент ожидает, что к аргументу `char const*` будет применено определяемое пользователем преобразование в `std::string`, и этот временный правосторонний (`rvalue`) аргумент будет передан в виде ссылки на константу в третьем параметре второй функции.

Однако, как отмечалось выше, стандартные преобразования имеют приоритет перед определяемыми пользователем. В предыдущем разделе, описывая преобразования, мы выяснили, что имеется стандартное преобразование из указателя в логическое значение. Если вы когда-либо рассматривали старый код, передающий простые указатели между функциями, то наверняка видели такие конструкции:

```
if (ptr) {  
    ptr->do_thing();  
}
```

Условное выражение в операторе `if` является указателем, а не логическим значением, но указатель может быть преобразован в `false`, если он равен нулю. Это более краткий идиоматический способ записи:

```
if (ptr != 0) {  
    ptr->do_thing();  
}
```

В современном C++ мы все реже видим простые указатели, тем не менее следует помнить, что это совершенно нормальное и разумное преобразование. Именно это стандартное преобразование компилятор считает более предпочтительным и выберет его вместо, казалось бы, очевидного определяемого пользователем преобразования из `char const*` в `std::string const&`. К удивлению клиента, компилятор вызовет перегруженную версию, которая принимает логическое значение в третьем аргументе.

Чья это ошибка? Ваша или клиента? Если бы клиент записал вызов так:

```
auto eh_office = make_office(24000.f, 200, "Eagle Heights"s);
```

ошибка не возникла бы. Литеральный суффикс сигнализирует о том, что этот объект на самом деле является объектом `std::string`, а не `char const*`. Так что в этом случае явно виноват клиент. Он должен знать о правилах преобразования.

Однако это не оправдывает выбранное вами решение. Вы должны реализовать интерфейс так, чтобы его проще было использовать правильно, чем неправильно. Пропустить литературный суффикс и тем самым допустить ошибку очень легко. Также подумайте, что случится, если перегруженная версия функции, принимающая логическое значение, будет добавлена *после* определения конструктора, принимающего `std::string const&`. До этого момента клиентский код будет действовать в соответствии с ожиданиями и с литературным суффиксом, и без него. Но после добавления перегруженной версии компилятор начнет выбирать лучшее преобразование и клиентский код может неожиданно начать действовать не так, как ожидалось.

Кто-то может посчитать этот пример неубедительным и попробовать заменить `bool` на более подходящий тип, например определить перечисление для использования вместо логического значения:

```
enum class floors {one, two};
office make_office(float floor_space, int staff,
    floors floor_count = floors::one);
office make_office(float floor_space, int staff,
    std::string const& building_name);
```

К сожалению, и этот подход не является выходом из спорной ситуации. Был введен новый тип, только чтобы способствовать правильному использованию набора перегруженных функций. Спросите себя, действительно ли такое решение выглядит яснее этого:

```
office make_office(float floor_space,int staff,bool two_floors = false,
    std::string const& building_name = {});
```

Если и этот довод показался вам неубедительным, то спросите себя, что вы будете делать, когда наступит следующий этап расширения предметной области и произвучит просьба-замечание: «Вообще-то, мы хотели бы иметь возможность давать названия и двухэтажным зданиям».

Вы должны реализовать интерфейс так, чтобы его проще было использовать правильно, чем неправильно.

ОДНОЗНАЧНАЯ ПРИРОДА АРГУМЕНТОВ ПО УМОЛЧАНИЮ

Преимущество аргумента по умолчанию в том, что любое преобразование сразу становится очевидным. Вы можете видеть, что `char const*` преобразуется в `std::string const&`. Нет никакой двусмысленности в выборе преобразования, потому что оно может произойти только в одном месте.

Кроме того, как упоминалось выше, наличие единственной функции дает больше уверенности, чем перегруженный набор. Если вы подобрали хорошее имя для своей функции и хорошо спроектировали ее, то вашему клиенту не придется задумываться о том, какую версию вызвать. Но, как показывает пример, это проще сказать, чем сделать. Аргумент по умолчанию сообщает клиенту, что функция обладает гибкостью, предоставляет альтернативный интерфейс к своей реализации и гарантирует единство семантики.

Единственная функция также позволяет избежать дублирования кода. Создавая перегруженную версию, вы исходите из самых лучших побуждений. Конечно, это так. Но перегруженные версии действуют немного по-разному, и вы решаете инкапсулировать оставшееся сходство кодов в одной функции, которую вызывают обе перегруженные версии. Однако со временем перегруженные версии начинают во многом перекрываться, потому что становится все труднее отделить фактические различия их применения. В конечном итоге вы столкнетесь с проблемой усложнения сопровождения базы кода по мере разрастания функционала.

Есть одно ограничение. Аргументы по умолчанию должны определяться в обратном порядке по списку параметров. Например, такое объявление будет допустимым:

```
office make_office(float floor_space, int staff, bool two_floors,  
                    std::string const& building_name = {});
```

А это — недопустимое:

```
office make_office(float floor_space, int staff, bool two_floors = false,
                   std::string const& building_name);
```

Если последнюю функцию вызвать только с тремя аргументами, то становится невозможно однозначно сказать что-либо о последнем аргументе, с каким параметром он должен быть связан: с `two_floors` или `building_name`?

Надеемся, мы смогли убедить вас, что перегрузку функций, несмотря на ее неоспоримые достоинства, не следует воспринимать поверхностно. Мы лишь слегка коснулись проблем разрешения перегрузки. Есть еще множество тонкостей, которые нужно изучить, если вы хотите по-настоящему понять, какая из перегруженных версий будет выбрана. Обратите внимание, что мы не рассматривали последовательности преобразований с много-точием и не обсуждали, что произойдет, если добавить в описанную схему шаблонную функцию. Однако если вы абсолютно уверены в необходимости использовать перегруженные версии, то мы вас убедительно просим: пожалуйста, не смешивайте аргументы по умолчанию с перегруженными функциями. Такая смесь трудно поддается анализу и расставляет ловушки для неосторожных. Это не тот стиль определения интерфейса, который проще использовать правильно, чем неправильно.

АЛЬТЕРНАТИВЫ ПЕРЕГРУЗКЕ

Перегрузка функций сигнализирует клиенту, что доступ к части функциональности, абстракции, можно обеспечить несколькими способами. Функции с одним и тем же идентификатором можно вызвать с разными наборами аргументов. На самом деле, вопреки ожиданиям, фундаментальный строительный блок API был описан как набор перегруженных функций, а не как функция.

Однако в несколько надуманном примере для этой главы можно обнаружить, что набор перегруженных функций:

```
office make_office(float floor_space, int staff, floors floor_count);
office make_office(float floor_space, int staff,
                   std::string const& building_name);
```

не так очевиден, как набор отдельных функций:

```
office make_office_by_floor_count(float floor_space, int staff,
                                    floors floor_count);
office make_office_by_building_name(float floor_space, int staff,
                                     std::string const& building_name);
```

Перегрузка функций — хороший инструмент, но его следует использовать с осторожностью. Это классный молоток, но им нельзя почистить апельсин. Идентификаторы, определяемые вами, должны указываться как можно точнее.

О перегрузке можно рассказывать очень долго — например, для выбора наилучшей из перегруженных версий процесс ранжирования имеет довольно длинный список тай-брейков; если бы это был учебник, мы бы подробно описали их все. Но в данном случае достаточно предупредить, что к перегрузке нельзя относиться легкомысленно.

ИНОГДА БЕЗ ПЕРЕГРУЗКИ НЕ ОБОЙТИСЬ

Описываемая рекомендация начинается словами: «Если есть выбор». Иногда может не быть возможности определить функцию с другим именем.

Например, может быть только один идентификатор конструктора. Поэтому если потребуется дать возможность создавать экземпляры класса несколькими способами, то вам действительно придется реализовать перегруженные версии конструктора.

Точно так же и операторы могут иметь единственное значение, очень ценное для ваших клиентов. Если по какой-то причине вы написали свой класс строк, то ваши клиенты предпочтут объединять строки таким способом:

```
new_string = string1 + string2;
```

а не:

```
new_string = concatenate(string1, string2);
```

То же верно в отношении операторов сравнения. Однако маловероятно, что при перегрузке операторов вам понадобится аргумент по умолчанию.

Стандарт предоставляет точку настройки `std::swap` и ожидает, что вы напишете перегруженную версию этой функции, оптимальную для вашего класса. В Core Guidelines имеется рекомендация «C.83. Для типов-значений желательно определить функцию `swap` со спецификатором `noexcept`», а она прямо предлагает создать перегруженную функцию. Однако и в этом случае крайне маловероятно, что при перегрузке функции понадобится аргумент по умолчанию.

Конечно, иногда просто нет доступных аргументов по умолчанию.

Итак, если вы *должны* выполнить перегрузку, делайте это сознательно и, повторим еще раз, *не* смешивайте аргументы по умолчанию с перегрузкой. В проектировании API это сравнимо с жонглированием заведенной бензопилой.

ПОДВЕДЕМ ИТОГ

Мы рассмотрели, как влияет рост базы кода на архитектуру API, исследовали простой пример перегрузки и увидели, что именно при этом может пойти не так. В главе, посвященной тонкостям перегрузки, мы кратко пробежались по правилам выбора перегруженной версии компилятором. С учетом работы по этим правилам мы показали, что может состояться вызов совсем не той из перегруженных версий, которая ожидалась. В частности, ошибка в нашем примере была вызвана предоставлением логического параметра с аргументом по умолчанию в перегруженной версии, что открывает широкие возможности для преобразования других нелогических аргументов в этот параметр. Нашей целью было показать, что аргументы по умолчанию предпочтительнее перегрузки функций и смешивание перегрузки с аргументами по умолчанию — весьма рискованное предприятие.

Пример, конечно же, был так себе, но факт остается фактом: для неосторожного инженера перегрузка таит серьезную опасность. Избежать опасности или минимизировать ее можно, разумно использовав аргументы по умолчанию и отказавшись от перегрузки. Желающие могут изучить все последствия перегрузки на своем любимом онлайн-ресурсе. Советуем сделать это, если когда-нибудь вы решите проигнорировать нашу вполне конкретную рекомендацию.

ГЛАВА 1.3

С.45. Не определяйте конструктор по умолчанию, который просто инициализирует переменные-члены; для этой цели лучше использовать внутриклассовые инициализаторы членов

ЗАЧЕМ НУЖНЫ КОНСТРУКТОРЫ ПО УМОЛЧАНИЮ

Начнем эту главу с краткого обзора. Рассмотрим рекомендацию из Core Guideline «NR.5. Не используйте двухфазную инициализацию». Она относится к привычке вызывать функцию инициализации после создания объекта. Эта практика зародилась еще в прошлом веке, когда язык С был самым актуальным и нужно было сначала разместить объект в стеке или в динамической памяти, а затем инициализировать его. Наиболее опытные программисты определяли функцию, которая принимала указатель на структуру в памяти, и давали ей имя `my_struct_init` или похожее.

Процесс состоял из двух фаз: размещения в памяти и последующей инициализации. Здесь что угодно может пойти не так: вы можете вставлять все больше и больше кода, выполняющегося между размещением в памяти

и инициализацией, и вдруг обнаружить, что использовали объект до инициализации.

Затем появились C++ и конструкторы, и эта проблема исчезла навсегда.

Для объектов, размещаемых статически, компоновщик создаст список их конструкторов, выполняемый до `main()`, и функцию для обхода этого списка. Компоновщик имеет полное представление о том, сколько места будут занимать эти объекты, поэтому функция сможет выделить память для них, инициализировать их все, а затем вызвать `main()`.

Для автоматических объектов компилятор выделяет некоторое пространство в стеке и инициализирует их в этой памяти. Для объектов, размещаемых в динамической памяти, оператор `new` вызовет оператор `new` для выделения памяти и конструктор — для инициализации объекта в этой памяти. Объекты, локальные для потоков выполнения, поддержка которых появилась в C++11, ведут себя почти так же, как статически размещаемые объекты, за исключением того, что их экземпляры создаются для каждого потока выполнения, а не для каждого экземпляра программы.

Мы надеемся, что вы уяснили четкую и последовательную схему, избавляющую от целого класса ошибок, связанных с использованием объекта до его готовности к этому. Объединение размещения в памяти и инициализации в одну операцию избавило от проблемы двухфазной инициализации.

Однако проблема не исчезла полностью и насовсем. У инженеров сохранилась привычка сначала создавать экземпляры объектов, а затем модифицировать их. Классы разрабатываются со значениями по умолчанию, и клиенты затем должны корректировать эти значения в соответствии с контекстом.

Эта привычка просто сместила проблему в другое место. Конструктор по умолчанию подходит не для всех классов. К сожалению, долгое время контейнеры, предоставляемые некоторыми реализациями C++, не работали с данными, не имеющими конструкторов по умолчанию. Конструктор по умолчанию в таких случаях предоставляется не как часть предметной области, а как часть области решения. Конечно, это означает возможность его использования и в предметной области, что вносит еще большую неразбериху в вопрос о правильном использовании класса.

Некоторые классы по своей природе должны иметь конструктор по умолчанию. Например, представьте объявление пустой строки. Для этой операции нет значимого API, если только вы не решите добавить перегруженную

версию конструктора специально для пустых строк со специальным параметром. API `std::string` распознает этот случай и предоставляет конструктор по умолчанию, создающий строку с нулевой длиной. Конструктор по умолчанию является очевидным решением. Действительно, все стандартные контейнеры предоставляют конструкторы по умолчанию.

Но не думайте, что ваш класс должен в обязательном порядке создаваться с помощью конструктора по умолчанию. Убедитесь, что знаете обо всех опасностях, прежде чем разрешить пользователям создавать экземпляры вашего класса без какой-либо спецификации.

КАК ИНИЦИАЛИЗИРУЮТСЯ ПЕРЕМЕННЫЕ-ЧЛЕНЫ

Вернемся к основной теме этой главы и рассмотрим процесс инициализации.

При создании объекта сначала резервируется память в соответствии с классом хранения, а затем вызывается конструктор. Однако для объектов встраиваемых типов правила немного отличаются. Если конструктор не определен, то члены класса инициализируются значениями по умолчанию. Если имеются члены встраиваемых типов, то они по умолчанию не инициализируются.

Это плохо: если не гарантировать инициализацию каждого члена класса, есть риск получить недетерминированное поведение программы. В таких случаях остается только пожелать вам удачи в отладке. Много лет назад один из авторов работал над игрой и использовал динамическое хранение данных, реализованное на C++. Игра поставлялась в виде двух запускаемых библиотек: одна для разработки и одна для распространения. Версия библиотеки для разработки была собрана с неопределенным макросом `NDEBUG`, вследствие чего срабатывали дополнительные проверки, и с помощью стандартной библиотеки можно было получить всевозможную отладочную информацию. При вызове оператора `new` память инициализировалась значением `0xcd`. Когда вызывался оператор `delete`, он заполнял освобождаемый блок памяти значением `0xdd`. Эта особенность позволяла выявлять попытки обращения к висячим указателям. Из соображений производительности библиотека, предназначенная для распространения, не делала этого, оставляя память как есть после выделения и освобождения.

Игра была многопользовательской. Компьютер каждого игрока отправлял свои ходы через интернет в мгновение ока, и все компьютеры должны были интерпретировать их одинаково. Для этого все компьютеры должны были находиться в идентичном состоянии с точки зрения модели игры; иначе модели игры на отдельных компьютерах оказывались в особой ситуации, и это приводило бы к рассогласованию результатов и невозможности продолжать игру. Такие несоответствия редко проявлялись в версиях игры, использующих библиотеку для разработки, потому что все они получали блоки выделенной памяти, заполненные значениями `0xcd`. Сбои наблюдались только в версиях с библиотекой, предназначеннной для распространения, и их было невероятно сложно отладить, потому что любые рассогласования в работе модели не замечались игроками, пока эти расхождения не начинали проявлять себя явно, а это происходило через достаточно большой промежуток времени после их возникновения.

До этой ситуации было невероятно трудно убедить команду разработчиков игры в важности инициализации каждой переменной-члена в каждом конструкторе. Когда наконец это дошло до всех ее членов, проблема исчезла. Детерминизм — ваш союзник, когда дело доходит до отладки, поэтому обеспечьте детерминизм, реализовав предсказуемое конструирование всех объектов, и инициализируйте все переменные-члены без исключения.

Есть три места, где можно инициализировать переменные-члены. Первое место — это тело функции-конструктора. Рассмотрим следующий класс:

```
class piano
{
public:
    piano();

private:
    int number_of_keys;
    bool mechanical;
    std::string manufacturer;
};
```

Вот как можно определить конструктор этого класса:

```
piano::piano()
{
    number_of_keys = 88;
    mechanical = true;
    manufacturer = "Yamaha";
}
```

Это вполне адекватное определение. Конструктор инициализирует все члены, причем в порядке их объявления. Здесь представлена инициализация в теле функции. Однако это не самый оптимальный подход. Перед выполнением тела функции-конструктора члены класса были инициализированы значениями по умолчанию. Сначала вызывался конструктор по умолчанию `std::string`, а затем — оператор присваивания с `char const*`. Фактически наш конструктор переопределяет значения в членах, а не инициализирует их.

Современные компиляторы достаточно интеллектуальны, чтобы заметить шаблон присваивания значений в конструкторе и оптимизировать его. `std::string` — это шаблонный класс, и высока вероятность, что все его выполнение доступно для компилятора. Он увидит избыточность инициализации и уберет ее. Однако такое поведение поддерживается не для всех классов. Поэтому желательно производить инициализацию с использованием списка инициализаторов, а не в теле функции.

Вот как можно изменить конструктор для этого:

```
piano::piano()
: number_of_keys(88)
, mechanical(true)
, manufacturer("Yamaha")
{}
```

Такой подход требует от инженера помнить о необходимости поддержки конструктора по умолчанию при добавлении переменных-членов, и код очень похож на шаблонный, который просто увеличивает размер файла.

Есть и третье место, где можно определить значения по умолчанию. Оно находится еще ближе к месту действия: в определении самого класса. Инициализаторы определяют значения по умолчанию для переменных-членов объекта, если в конструкторе не указано другое. Вернемся к определению нашего класса, чтобы посмотреть, как определить эти инициализаторы:

```
class piano
{
public:
    // piano(); // больше не нужен

private:
    int number_of_keys = 88;
    bool mechanical = true;
    std::string manufacturer = "Yamaha";
};
```

Так намного лучше. В этом примере мы избавились от лишней функции-члена и определили характеристики фортепиано по умолчанию: 88-клавишное механическое пианино фирмы Yamaha. Такое решение имеет свою цену, которую нельзя игнорировать, а именно: эти значения по умолчанию экспортируются в составе объявления класса, от которого, возможно, будет зависеть множество других файлов с исходным кодом. Изменение любого из этих значений может потребовать повторной компиляции неизвестного количества файлов. Однако есть веские причины считать эту цену оправданной.

ЧТО МОЖЕТ СЛУЧИТЬСЯ, ЕСЛИ ПОДДЕРЖИВАТЬ КЛАСС БУДУТ ДВА ЧЕЛОВЕКА

Обычно класс поддерживается одним человеком. Этот инженер определяет абстракцию, реализует ее в классе, проектирует API и имеет полное представление о происходящем.

Конечно, всякое случается. Человека, поддерживавшего класс, могут на время перевести в другой проект, или, что еще хуже, он может уйти совершенно внезапно, не выполнив надлежащую передачу. Многое может осложниться без строгой дисциплины передачи информации через документацию, встречи и другие мероприятия, которые типичному инженеру кажутся досадной тратой времени.

Сборная солянка из конструкторов

Когда над классом работают несколько человек, начинают возникать несоответствия. Большая часть Core Guidelines посвящена уменьшению вероятности возникновения несоответствий. Непротиворечивый код проще читать, он содержит меньше сюрпризов. Подумайте, что могло бы случиться с классом `piano`, если бы на него набросились сразу три сопровождающих разработчика:

```
class piano
{
public:
    piano()
        : number_of_keys(88)
        , mechanical(true)
        , manufacturer("Yamaha")
```

```
{}
piano(int number_of_keys_, bool mechanical_,
       std::string manufacturer_ = "Yamaha")
: number_of_keys(number_of_keys_)
, mechanical(mechanical_)
, manufacturer(std::move(manufacturer_))
{}
piano(int number_of_keys_) {
    number_of_keys = number_of_keys_;
    mechanical = false;
    manufacturer = "";
}

private:
    int number_of_keys;
    bool mechanical;
    std::string manufacturer;
};
```

Это лишь пример, но подчас приходится видеть подобные вещи в реальности. Обычно конструкторы отделяются друг от друга множеством строк. Все конструкторы могут быть определены в определении класса. Поэтому не сразу видно, что существует три очень похожих конструктора: они прячутся за большим количеством строк реализации. По имеющемуся коду порой можно даже кое-что рассказать о разных сопровождающих програмистах. Разработчик третьего конструктора, похоже, не знает о списках инициализации. Кроме того, присваивание пустой строки члену `manufacturer` является избыточным, из чего можно сделать вывод, что разработчик не знает, как работают конструкторы и инициализация по умолчанию.

Что еще более важно, первый и третий конструкторы присваивают переменным-членам разные значения по умолчанию. Подобное обстоятельство можно заметить и проследить в этом простом примере, но представьте не столь очевидную ситуацию. Вызывающий код может передавать один, два или три аргумента и в разных случаях получать разное поведение, чего не хочет ни один пользователь. Наличие аргументов по умолчанию в перегруженных конструкторах тоже должно вызывать беспокойство.

Что произойдет, если применить внутриклассовые инициализаторы членов? Ниже размещен пример такого кода:

```
class piano
{
public:
    piano() = default;
```

```
piano(int number_of_keys_, bool mechanical_, std::string manufacturer_)  
    : number_of_keys(number_of_keys_)  
    , mechanical(mechanical_)  
    , manufacturer(manufacturer_)  
{}  
piano(int number_of_keys_) {  
    number_of_keys = number_of_keys_;  
}  
  
private:  
    int number_of_keys = 88;  
    bool mechanical = true;  
    std::string manufacturer = "Yamaha";  
};
```

Теперь все значения по умолчанию приведены в соответствие. Наличие внутриклассовых инициализаторов членов сообщило авторам конструкто-ров, что значения по умолчанию уже выбраны и им не нужно и не следует присваивать свои значения.

Аргументы по умолчанию могут запутать ситуацию в перегруженных функциях

Аргументы по умолчанию в конструкторах сбивают с толку. Они подразумевают значение по умолчанию для чего-либо, но между этим значением по умолчанию и объявлением переменной-члена существует когнитивная дистанция. Можно придумать правдоподобные причины для добавления в конструктор параметра по умолчанию, например: в класс был добавлен новый член и, чтобы не подвергать изменению весь клиентский код, вы решили взять на себя технический долг и определить значение по умолчанию. Однако этот технический долг необходимо погасить: в идеале — добавить дополнительный конструктор, который принимает все требуемые параметры, и объявить устаревшим прежний конструктор.

ПОДВЕДЕМ ИТОГ

Выбор определять или не определять конструкторы по умолчанию должен делаться осознанно. Не все классы имеют осмысленные значения по умолчанию. Инициализация переменных-членов может происходить в трех местах: в теле функции-конструктора, в списке инициализации конструктора и в объявлении переменных-членов, известных как инициализаторы членов по умолчанию.

Инициализаторы членов по умолчанию определяют значения по умолчанию в точке объявления. Если есть член, который нельзя определить таким способом, то может не быть и легального механизма, посредством которого определяется конструктор по умолчанию. Это нормальное явление. Как отмечалось выше, в конструкторах по умолчанию вообще нет особой необходимости.

Конструкторы создают вариант по умолчанию. Определение инициализаторов переменных-членов со значениями по умолчанию делает каждый вариант более специфичным. В этом дополнительная польза для клиента, так как он может точнее и конкретнее настроить объект под свои требования и не испытывать беспокойства по поводу его состояния.

ГЛАВА 1.4

С.131. Избегайте тривиальных геттеров и сеттеров

https://t.me/it_boooks

АРХАИЧНАЯ ИДИОМА

Тривиальные геттеры (методы чтения свойства) и сеттеры (методы записи в свойство) — пережиток ранних этапов развития C++. Обычно они выглядят так:

```
class x_wrapper
{
public:
    explicit x_wrapper(int x_) : x(x_) {}
    int get_x() const { return x; } // это геттер
    void set_x(int x_) { x = x_; } // это сеттер

private:
    int x;
};
```

Функции `get` и `set` просто обращаются к свойству класса и возвращают или изменяют его значение. На первый взгляд, этот подход имеет определенные преимущества. Можно поискать в коде имена `get_x` и `set_x`, чтобы увидеть, где изменяется или извлекается значение свойства `x`. Также можно установить точку останова в функциях и в отладчике перехватить все экземпляры, где извлечено или изменено значение. Этот подход отвечает требованиям сохранения конфиденциальности данных: данные инкапсулируются за API.

Но эти функции тривиальны. Они просто препятствуют прямому доступу к `x`. Core Guideline не рекомендует такой подход. Однако из примера бесполезного класса `x_wrapper` трудно понять, почему это не самое лучшее

решение. Ниже мы приведем несколько более реалистичных примеров. Но пока поговорим об абстракции, зачем она нужна и почему так важна в программировании на C++. А попутно рассмотрим немного истории, познакомим вас с инвариантами классов и покажем важность правильно-го выбора существительных и глаголов для генерации идентификаторов функций-членов.

АБСТРАКЦИИ

Одна из основных задач языка программирования — помочь в определении и воплощении абстракций. Итак, что фактически мы делаем, когда опреде-ляем абстракции с помощью языка программирования C++?

Мы трансформируем фрагменты памяти с числами в представления объ-ектов нашей предметной области. Это одна из самых сильных сторон C++, а также одна из его основных целей.

Дом можно представить как набор чисел. Они могут определять размеры земельного участка, на котором тот построен; высоту; площади помещений; количество этажей, комнат, окон и чердачков.

Такая абстракция реализована как последовательность чисел, или полей, объединенных в запись. Запись определяет порядок расположения по-лей, а это означает, что, имея набор записей, последовательно хранящихся в памяти, можно легко и просто перейти к любой записи в наборе и из-влечь любое поле из этой записи с помощью простых арифметических действий.

На заре развития вычислительной техники много времени уходило на работу с наборами таких записей или таблиц. Простая обработка данных включала сбор некоторых данных и создание записей или чтение некоторых записей и создание дополнительных данных. Жизнь была простой, и мы были счастливы.

Если вы хоть немного знакомы с C++, то без труда узнаете структуры в записях, переменные-члены в полях и массивы структур в таблицах. Для простой обработки данных представляется разумным напрямую читать и изменять поля в записях.

Но представьте, что требуется нечто большее, чем перебор записей. До по-явления классов приходилось хранить данные в структурах и вызывать

функции для работы с данными в них. До появления конструкторов копирования и операторов присваивания не было никакой возможности передавать экземпляры структур функциям, только указатели на них. До появления уровней доступа можно было напрямую изменять любые поля в структурах, поэтому было очень важно знать, что делается с данными и что все остальные ожидают от этих данных. Порой это вызывало значительные умственные нагрузки.

Примечательно, что обычно данные, находящиеся за пределами области видимости выполняемой в данный момент функции, были напрямую доступны для модификации. Они могли совместно использоваться несколькими функциями путем помещения их в область видимости, что позволяло одной функции записывать данные, а другим — читать их. Вот как это выглядело:

```
int modifying_factor;

void prepare_new_environment_data(int input)
{
    int result = 0;
    /* ... подготовить новое значение данных на основе аргумента input */
    modifying_factor = result;
}

int f2(int num)
{
    int calculation = 0;
    /* ... выполнить некоторые вычисления */
    return calculation * modifying_factor;
}
```

Если инженеру выпадало счастье быть единственным человеком, изменяющим такие данные, он мог рассуждать об изменениях с определенной степенью уверенности. Он знал, где изменяются данные и где они извлекаются, поэтому мог быстро находить причины ошибок, связанных с данными.

К сожалению, с увеличением численности команд и важности отдельных частей данных нести ответственность за информацию становилось все труднее. Данные объявлялись в заголовочных файлах, что делало их доступными для многих инженеров. Рассуждения о данных становились все более безнадежным делом. Приходилось писать пространные инструкции, регламентирующие использование данных, вспыхивали жаркие споры о том, кому принадлежат эти данные, созывались совещания, чтобы раз и навсегда установить правила, определяющие, кому разрешено изменять данные и при каких условиях. К правилам добавлялась документация,

и все надеялись, что ее обновит кто-то другой. Каждый считал, что другие инженеры рассуждают о данных точно так же, как он. Каждый верил, что через полгода все остальные инженеры продолжат рассуждать о данных так же, как делали это прежде.

Важно помнить, что, как только данные объявляются в заголовочном файле, теряется не только контроль над ними, но и всякая надежда на контроль, и что бы вы ни имели в виду под этим фрагментом данных, начинает безжалостно действовать закон Хайрама: «Если число пользователей API достаточно велико, то неважно, что вы обещаете в контракте: любое наблюдаемое поведение системы будет зависеть от чьих-то действий»¹.

Вы больше не владеете данными, но при этом несете за них ответственность. Поиск в базе кода идентификатора с полезным и, следовательно, общепринятым именем превращается в кошмар оценки контекста.

В конце концов способные инженеры научились прятать часть данных за парой функций, одной для записи, а другой для чтения. Это не упрощало рассуждений о данных, но поиск в коде мест вызова этих функций вкупе с точками останова в них позволял выявлять, при каких обстоятельствах изменяются данные.

А вот *самые* способные инженеры давали функциям говорящие имена, отражающие их назначение в предметной области, стремясь писать самодокументирующийся код. Например, если часть данных представляла высоту над уровнем моря, они могли назвать функции следующим образом:

```
/* elevation.h */
void change_height(int new_height);
int height();

/* elevation.c */
int height_ = 0;

void change_height(int new_height)
{
    height_ = new_height;
}

int height()
{
    return height_;
}
```

¹ <https://www.hyrumslaw.com>

Такие имена не мешали другим произвольно менять высоту в неподходящие моменты в своем коде, но по крайней мере сообщали о том, что делают эти функции, и, возможно, улучшали читабельность кода за счет использования существительных в именах функций, возвращающих значения, и глаголов — в именах функций, изменяющих значения. Такой прием защиты данных позволял находить в коде места, где они используются, и давал дополнительный бонус в виде возможности устанавливать точки останова в отладчике, чтобы внимательно отследить все попытки.

ПРОСТАЯ ИНКАПСУЛЯЦИЯ

Вместе с классами появились уровни доступа и функции-члены. О! Уровни доступа! Наконец данные принадлежат мне. Я могу сделать их своей собственностью, объявив приватными, и тогда никто не сможет коснуться их, кроме как через одну из общедоступных функций-членов. О! Функции-члены! В С можно было хранить указатели на произвольные функции в структурах, но функции-члены с неявным указателем на экземпляр в первом параметре стали настоящим прорывом, настолько просто их было использовать. Мир заиграл новыми красками.

Стиль программирования, заключающийся в сокрытии данных за парой функций, перестал быть просто блестящей идеей: он превратился в обычную практику, реализуемую на уровне языка. Был предложен новый совет, четкий и ясный: «Сделай все свои данные приватными». Было добавлено новое ключевое слово `class`, отличающее классы от структур `struct`. Для обратной совместимости члены структур по умолчанию оставались общедоступными, а члены классов — приватными (закрытыми). Так сохранялась и поддерживалась идея о том, что данные должны быть закрытыми.

Вместе с возможностью хранить данные в приватном интерфейсе появилась возможность предотвратить постороннее вмешательство в эти данные. Вы можете создать объект и предоставить функции `get/set` для доступа к данным. Это и есть инкапсуляция.

Инкапсуляция — один из основных принципов объектно-ориентированного программирования: ограничение доступа к частям объекта и объединение данных с функциями для их обработки. Это своеобразный механизм защиты, помогающий инженерам и пользователям ясно представлять

причинно-следственные связи. Внезапно инкапсуляция стала доступна программистам на C, и произошло следующее.

До:

```
struct house
{
    int plot_size[2];
    int floor_area;
    int floor_count;
    int room_count;
    int window_count;
};
```

После:

```
class house
{
public:
    int get_plot_x() const;
    int get_plot_y() const;
    int get_floor_area() const;
    int get_floor_count() const;
    int get_room_count() const;
    int get_window_count() const;

    void set_plot_size(int x, int y);
    void set_floor_area(int area);
    void set_floor_count(int floor_count);
    void set_room_count(int room_count);
    void set_window_count(int window_count);

private:
    int plot_size[2];
    int floor_area;
    int floor_count;
    int room_count;
    int window_count;
};
```

Одновременно появились стандарты программирования, требующие, например, чтобы «все данные были закрытыми и имели собственные методы чтения и записи». Кое-где даже предписывалось снабжать функции `get` квалификатором `const`. Залитые солнцем долины кода, свободного от ошибок, лежали прямо за следующим холмом. И все напитки были за счет заведения.

Но постойте-ка, а что в действительности мы получили? Проблемы и не приятности, которые преследовали нас раньше, никуда не исчезли. Мы все

еще вынуждены беспокоиться о возможности изменения данных другими. В чем же тогда смысл функций получения данных?

Вот мы и добрались до сути данной рекомендации. Не поступайте так. Не пишите геттеры и сеттеры, которые не делают ничего, кроме пересылки данных между точкой вызова и объектом. Нет смысла делать данные приватными, если вы просто решили открыть к ним доступ другими способами. Для этого достаточно сделать данные общедоступными. Пряча данные за функциями `get` и `set`, вы ничего не добавляете к пониманию вашего класса пользователями. Просто получая или изменяя данные с помощью методов, вы никак не используете всю мощь C++. Хуже того, вы заменили компактный, хотя и потенциально опасный API `struct house` с общедоступными членами раздутым API `class house`. Его труднее охватить одним взглядом. Кроме того, по-прежнему остается неясной семантика изменения количества этажей. Такое решение выглядит очень странно.

Функции `get` и `set` не делают интерфейс более безопасным или менее подверженным ошибкам. Их использование — просто еще один путь для проникновения ошибок.

Вы должны задать себе вопрос: *почему* вас беспокоит возможность изменения данных другими? Станут ли недействительными экземпляры вашего класса, если произойдет произвольное изменение переменных-членов? Наложены ли какие-то ограничения на ваши данные? Если да, то ваша функция `set` должна выполнять какие-то дополнительные действия. Например, обеспечить выполнение условия, что площадь пола не должна превышать площадь участка:

```
void house::set_floor_area(int area)
{
    floor_area = area;
    assert(floor_area < plot_size[0] * plot_size[1]);
}
```

**Нет смысла делать
данные приватными,
если вы просто решили
открыть к ним доступ
другими способами.**

Теперь метод записи в свойство перестал быть тривиальным, и появилась веская причина его существования.

Рассмотрим другой пример: банковский счет. Вот неправильный способ его реализации:

```
class account
{
public:
    void set_balance(int);1
    int get_balance() const;

private:
    int balance;
};
```

Более удачное решение содержит бизнес-логику:

```
class account
{
public:
    void deposit(int);
    void withdraw(int);
    int balance() const;

private:
    int balance_;
};
```

Теперь при изменении баланса будут выполняться действия, определяемые бизнес-логикой. Функция `deposit` увеличит баланс, а функция `withdraw` уменьшит его.

ИНВАРИАНТЫ КЛАССА

Условие, согласно которому площадь пола не может превышать размер участка, известно как инвариант класса. В более общем смысле инвариант — это условие, которое должно выполняться для всех действительных экземпляров класса. Условие задается при создании общедоступных функций-членов и поддерживается между их вызовами. Инвариантом для класса `account` может быть условие, согласно которому значение `balance` не должно опускаться ниже 0, или сумма всех поступлений и списаний должна быть равна балансу. Пример инварианта для класса `house`, согласно которому площадь пола не может превышать размер участка, выражается в функции `void house::set_floor_area(int area)`. Именно выражение этого инварианта класса делает метод записи в свойство нетривиальным.

¹ Никогда не используйте тип `float` в финансовых вычислениях. Ошибки округления и представления будут накапливаться и приводить к неправильным результатам.

Взгляните на следующее определение класса. Сможете ли вы самостоятельно найти инварианты?

```
class point
{
public:
    void set_x(float new_x) {
        x_ = new_x; }
    void set_y(float new_y) {
        y_ = new_y; }
    float get_x() const {
        return x_; }
    float get_y() const {
        return y_; }

private:
    float x_;
    float y_;
};
```

Не нашли? И правильно, потому что их здесь нет. Изменение координаты x не влияет на координату y . Они могут меняться независимо, а точка остается точкой. Функции-члены не добавляют ничего нового. Именно от возникновения подобной ситуации предостерегает данная рекомендация.

А вот еще один пример:

```
class team
{
public:
    void add_player(std::string name) {
        if (players_.size() < 11) players_.push_back(name); }
    std::vector<string> get_players() const {
        return players_; }

private:
    std::vector<string> players_;
}
```

Инвариант класса заключается в ограничении числа игроков в команде: их может быть не более 11.

Уровни доступа позволяют разработчику класса поддерживать инварианты класса, проверяя влияние операций на переменные-члены и убеждаясь, что все данные сохраняют допустимые значения.

В этой рекомендации приводится причина: «Тривиальный геттер или сеттер не добавляет семантической ценности; элемент данных с тем же

успехом может быть общедоступным». Как было показано в примерах с домом и с точкой, это именно так. Нет semanticской разницы между `struct house` и `class house`. Функции-члены в классе `point` избыточны. Существует, конечно, операционная разница: наличие методов доступа позволяет установить точки останова, чтобы выявить все места, где происходит чтение или изменение переменных-членов, но это не влияет на semanticическую природу абстракции дома или точки.

При создании класса и проектировании его интерфейса обычно думают об аспектах абстракции, моделируемой числами: в конце концов, программы пишутся для компьютера. Иногда, однако, возникает искушение сделать эти аспекты числовыми элементами данных. Если клиент захочет что-то изменить, он вызовет функцию, чтобы установить новое значение, и после некоторой проверки, связанной с поддержкой инвариантов, соответствующий элемент данных модифицируется должным образом.

СУЩЕСТВИТЕЛЬНЫЕ И ГЛАГОЛЫ

Но такой подход не учитывает назначение классов. Класс — это абстракция идеи, а не набор данных. Причина, объясняющая, почему `change_height` и `height` считаются хорошими идентификаторами функций, заключается в том, что они описывают происходящее в реальности. Инженера-заказчика волнует не факт изменения целочисленной величины, а факт изменения высоты объекта. Для него важно не значение члена `height_`, а опять же высота объекта. Это два разных уровня абстракции, и опускание деталей абстракции — плохая практика, излишне увеличивающая мыслительную нагрузку на заказчика из предметной области.

Приватные данные и реализация моделируют абстракцию как физическую сущность. Ожидается, что общедоступные функции-члены будут наполнены определенным смыслом, связанным с самой абстракцией-сущностью, а не с ее реализацией. Это точно не относится к функциям `get` и `set`. Разделение на общедоступное и приватное определяет различные уровни абстракции.

В частности, этот подход со всей очевидностью напоминает концепцию «модель — представление — контроллер» (Model-View-Controller, MVC). Представьте для примера старый ядерный реактор, состоящий из активной зоны и пульта управления со множеством кнопок, переключателей и ползунков, а также ряды и ряды индикаторов и стрелочных приборов. Индикаторы и приборы дают информацию о состоянии активной зоны; они — ваши

«глаза». Кнопки, переключатели и ползунки каким-то образом изменяют параметры работы активной зоны; они — ваши исполнительные механизмы (контроллеры).

То же можно видеть в API. Функции-члены со спецификатором `const` позволяют получить информацию об объекте. Другие функции-члены управляют объектом. Переменные-члены моделируют объект. При выборе идентификаторов функций представлений и контроллеров для первых имеет смысл выбирать имена существительные, а для вторых — глаголы, но не идентификаторы `get` и `set`.

Поскольку модель сконструирована из приватных переменных-членов, подобную реализацию можно обновлять по мере ее развития. Это сложно сделать, используя пары `get`/`set`: обновление реализации означает обновление кода во всех точках вызова функций `get`/`set`. Одно из существенных преимуществ правильно определенного общедоступного интерфейса — он гораздо стабильнее приватного. А это означает, что клиентский код вряд ли потребуется менять с изменением реализации абстракции.

После всего вышесказанного следует отметить, что иногда в программах нужно лишь передавать пакеты связанных данных и не утруждать себя созданием класса, в котором нет никакой необходимости, хотя бы просто потому, что данные могут быть всего лишь блоками байтов, прочитанных из файла или сетевого соединения. В таких случаях предпочтительнее использовать структуры. Тогда данные будут общедоступны, а геттеры и сеттеры окажутся не нужны и не важны.

ПОДВЕДЕМ ИТОГ

- Тривиальные геттеры и сеттеры не добавляют в интерфейс ничего полезного.
- Нетривиальные сеттеры должны гарантировать соблюдение инвариантов класса.
- Существительные и глаголы могут служить превосходными идентификаторами функций, лучше отражающими абстракцию.
- Несвязанные данные без инвариантов класса можно объединять в структуры.

ГЛАВА 1.5

ES.10. Объявляйте имена по одному в каждом объявлении

https://t.me/it_boooks

ПОЗВОЛЬТЕ ПРЕДСТАВИТЬ

Объявление вводит имя в программу. Оно также может ввести имя повторно: множественные объявления вполне допустимы, так как без них невозможны предварительные объявления. Определение — это особый вид объявления, оно содержит в себе достаточно деталей, чтобы дать возможность использовать то, что упомянуто в объявлении.

Существует на удивление большое количество типов объявлений. Например, функцию можно объявить так:

```
int fn(int a);
```

Это очень распространенный вид объявлений. Оно вводит имя `fn` типа «функция, принимающая и возвращающая целое число». Вы можете вызывать эту функцию в своем коде, не определяя ее заранее. Также можно объявить шаблон функции или класса, например:

```
template <class T, class U>
int fn(T a, U b);

template <class T>
class c1 {
public:
    c1(T);
private:
    T t;
};
```

Здесь объявлена функция, принимающая любые типы `T` и `U` и возвращающая `int`, а также класс, основанный на произвольном типе `T`. Аналогично

можно объявить частичные и явные специализации шаблонов, а также явные экземпляры шаблонов:

```
template <class T> int fn(T a, int b); // частичная специализация
template <> int fn(float a, int b); // явная специализация
template class cl<int>; // явный экземпляр
```

Можно объявить пространство имен:

```
namespace cg30 {
... // дополнительные объявления
}
```

Можно объявить внешние ссылки для взаимодействия с программами на других языках:

```
extern "C" int x;
```

Можно объявить атрибут:

```
// Предупреждать, если возвращаемое значение отбрасывается,
// а не присваивается
[[nodiscard]] int fn(int a);
```

Существует также набор объявлений, которые можно делать только внутри блока, например в теле функции. К ним относятся объявления `asm`, в которых можно указать явные инструкции на языке ассемблера для вставки в код, генерируемый компилятором:

```
asm {
    push rsi
    push rdi
    sub rsp, 184
}
```

Объявив псевдоним, можно создать более короткое и удобное в обращении имя типа:

```
using cl_i = cl<int>;
```

Аналогично можно объявить псевдоним для вложенного пространства имен:

```
namespace rv = std::ranges::views;
```

Объявления `using` позволяют вводить в текущий блок имена из других пространств имен:

```
using std::string // После этого объявление можно больше не печатать префикс
                  // std:: для обращения к типу string в текущем блоке
```

Директива `using`, в отличие от объявления `using`, позволяет внедрить в текущий блок пространство имен целиком:

```
using namespace std; // Никогда не поступайте так в глобальном
                    // пространстве имен. Никогда. Никогда, никогда.
                    // Ни для какого пространства имен.
```

Объявление `using enum` внедряет содержимое перечисления в текущий блок:

```
using enum country; // перечисление country может быть определено
                    // где-то еще
```

Объявление `static_assert` можно делать в блоке:

```
static_assert(sizeof(int) == 4); // Проверят, что тип int имеет размер
                                // четыре байта
```

Непрозрачные объявления `enum` позволяют объявлять перечисления без их определения, просто указав базовый тип и, следовательно, его размер:

```
enum struct country : short;
```

Можно даже объявить «ничто»:

```
;
```

Наконец, есть простые объявления.

Далее мы сосредоточимся на объявлении объектов. Смысл следующего объявления должен быть вам понятен:

```
int a;
```

Здесь создается экземпляр целочисленного типа с именем `a`. Пока ничего сложного. Конечно, этот экземпляр не инициализирован (если только не объявлен в глобальном пространстве имен), а его идентификатор не имеет большого значения, поэтому не будем заострять на нем наше внимание.

Следует отметить, что в подобных объявлениях стандарт позволяет указывать списки идентификаторов, разделенных запятыми. Каждый идентификатор интерпретируется как отдельное объявление с теми же спецификаторами. Например:

```
int a, b, c, d;
```

Здесь создаются четыре экземпляра целочисленного типа с именами `a`, `b`, `c` и `d`.

ОБРАТНАЯ СОВМЕСТИМОСТЬ

Взгляните на следующее объявление:

```
int* a, b, c, d;
```

Объекты какого типа оно создает?

Надеемся, вы заметили ловушку и сказали, что `a` — это указатель на `int`, а `b`, `c` и `d` — это экземпляры типа `int` (это одна из причин, почему при объявлении указателей звездочка часто указывается рядом с идентификатором, а не с типом). Конечно, вы заметили, потому что ждали подвоха, но иногда в практике приходится сталкиваться с ошибками компиляции, когда программист полагал, что все эти объекты являются указателями на `int`.

Такое объявление законно, но неразумно. Это ловушка для невнимательных, и ее следует исключить из языка. По крайней мере, таково мнение опытных разработчиков. К сожалению, такие объявления по стандарту языка считаются допустимыми. Причина — соображения обратной совместимости. Язык программирования C допускает это, потому что раньше требовал объявления всех объектов в начале функции. Эти объявления помогали компилятору определить, насколько должен вырасти стек, и вычислить адреса для всех данных в точке входа в функцию. Это была удобная оптимизация. Однако разумнее объявлять каждый объект в отдельной строке, например, так:

```
int* a;  
int b;  
int c;  
int d;
```

Обратную совместимость невозможно переоценить. Одна из фундаментальных причин успеха C++ заключалась в его способности компилировать миллионы строк уже существующего, созданного значительно ранее, кода. Когда вы создаете новый язык, такой как Rust или Go, никакого кода на нем не существует, разве только образцы, разработанные самим автором-изобретателем языка. Язык C++, напротив, позволяет выбирать части существующих программ и постепенно заменять их более четкими абстракциями. Например, техника позднего связывания функций была доступна в C с самого начала: вы могли сохранять указатели на функции в структурах и вызывать определенные члены структур в нужные моменты. C++ formalizовал этот подход и ввел ключевое слово `virtual`, сигнализирующее о том, что это было намерением программиста. Но при желании вы все еще можете делать что-то по-старому, не используя ключевое слово `virtual`:

```

using flump_fn_ptr = void()(int);

struct flumper {
    flump_fn_ptr flump;
    ...
};

void flump_fn(int);

void fn()
{
    flumper f1 { flump_fn; }
    f1.flump(7);
}

```

Существует крайне ограниченное количество случаев, когда может возникнуть потребность в использовании этого старого подхода вместо виртуальных функций, но он все еще поддерживается. Его применение не лучшая идея, но вы все-таки можете к нему прибегнуть.

Точно так же и в целом: несмотря на непрекращающееся развитие языка C++, сохраняется возможность скомпилировать и запустить старый код, за исключением некоторых конструкций, таких как `std::auto_ptr`, на замену которой пришел тип `std::unique_ptr`.

ПИШИТЕ БОЛЕЕ ЯСНЫЕ ОБЪЯВЛЕНИЯ

Вернемся к теме рассматриваемой рекомендации. Требование раннего объявления может привести к необходимости объявления довольно большого количества переменных в начале функции. В Core Guidelines представлен следующий пример:

```
char *p, c, a[7], *pp[7], **aa[10];
```

Если бы в одном объявлении не допускалось перечислять несколько имен, его можно было бы расширить и оно стало бы таким:

```
char *p; char c; char a[7]; char *pp[7]; char **aa[10];
```

или, что еще лучше, таким:

```

char *p;
char c;
char a[7];
char *pp[7];
char **aa[10];

```

Пять строк когнитивной недвижимости вместо одной.

«Но, — можете подумать вы, — что, если моей функции требуется много переменных? Что делать в таком случае? Перечислить каждую в своей строке и занять половину экрана? А если мне понадобится сгруппировать соответствующие переменные, то тогда для отделения групп друг от друга придется вставить еще и пустые строки. В таком случае я буду вынужден просматривать массу кода, прежде чем доберусь до фактического начала функции. Это ухудшит читабельность. Такая идея мне не нравится».

Ключевыми словами здесь являются «сгруппировать соответствующие переменные». Это напоминает термин «формирование абстракции». Если у вас есть переменные, связанные каким-то образом, то формализуйте эту связь, определив структуру, и назовите ее так, чтобы зафиксировать и идентифицировать эту связь. Добавьте необходимую функциональность в структуру. После этого вы сможете заменить несколько строк объявлений одним объявлением экземпляра этой структуры. Всегда будьте готовы создать полезную абстракцию.

СТРУКТУРНОЕ СВЯЗЫВАНИЕ

Из этого правила есть исключение, которое применяется, когда речь идет об использовании структурного связывания. Это нововведение появилось в C++17, оно позволяет связать массив или тип класса, не являющегося объединением, со списком имен. Вот пример простейшего случая:

```
int a[2] = {1, 2};
auto [x, y] = a; // x = a[0], y = a[1]
```

Аналогично выполняется связывание с типом кортежа:

```
std::unordered_map<std::string, int> dictionary;
auto [it, flag] = dictionary.insert({"one", 1});
// it – указывает на элемент
// flag – признак успешного выполнения операции вставки
```

Точно так же работает связывание со структурой для ее распаковки:

```
struct s {
    int a;
    float b;
};
```

```
s make_s();  
  
auto [sa, sb] = make_s(); // sa имеет тип int  
// sb имеет тип float
```

В каждом случае в одном объявлении упоминается несколько имен. Однако мы можем уменьшить любую путаницу, объявив имена в отдельных строках, например:

```
std::unordered_map<std::string, int> dictionary;  
auto [it, // it указывает на элемент  
    flag] // flag – признак успешного выполнения операции вставки  
= dictionary.insert({"one", 1});
```

То, что выглядит упрощением для одного, может оказаться усложнением для другого, поэтому выбор способа — это дело личного вкуса.

ПОДВЕДЕМ ИТОГ

Многие рекомендации в Core Guidelines способствуют написанию удобочитаемого и легко воспринимаемого кода. Текущая рекомендация касается старых привычек и сводится к трем соображениям.

- Существует много видов объявлений.
- Имена могут вводиться где угодно.
- Не запутывайте объявления, вводя более одного имени в строке, если этого не требует язык.

ГЛАВА 1.6

NR.2. Функции не обязательно должны иметь только один оператор возврата

ПРАВИЛА МЕНЯЮТСЯ

Меня¹ удивляет, что даже в XXI веке, вот уже 20 лет, люди все еще спорят об этом. Этот раздел Core Guidelines называется Non-rules and myths («Надуманные правила и мифы»). В него попадает удивительно распространенный совет, согласно которому функции должны иметь только один оператор возврата.

Требование единственности оператора возврата в функции — это старое-престарое правило. Как же легко забываются успехи, достигнутые в программировании. Я, вернее, мои родители купили мне первый компьютер в далеком 1981 году. Это был Sinclair ZX81 с процессором NEC Z80, работающим на частоте 3,25 МГц. Операционная система, вшитая в ПЗУ объемом 8 Кбайт, включала интерпретатор BASIC, позволяющий писать простые программы, которым был доступен скучный 1 Кбайт ОЗУ.

Мне было 14 лет, я хотел писать игры и обнаружил, что лучший способ для этого — полностью отказаться от интерпретатора BASIC и писать на языке ассемблера Z80. С помощью руководства *Mastering Machine Code on Your ZX81* Тони Бейкера (Toni Baker)², поразительной книги *Programming*

¹ Экскурс в историю. Один из авторов, Дж. Гай Дэвидсон, делится собственным опытом постижения этой рекомендации.

² Baker T. Mastering Machine Code on Your ZX81. — Reston, VA: Reston Publishing Company, Inc., 1982.

the Z80 Роднея Закса (Rodnay Zaks)¹ и ассемблера я смог написать и продать школьным друзьям свои первые игры.

Писать на ассемблере Z80 было гораздо труднее, чем на BASIC. В частности, в BASIC были номера строк и идея подпрограмм. Я мог выполнять ветвление алгоритма с помощью оператора `GOTO`, или же `GOSUB`, который вернет меня туда, где интерпретатор обработал ключевое слово `RETURN`. По мере накопления опыта программирования на ассемблере Z80 я стал замечать все больше общих концепций между ним и BASIC и понимать природу языков программирования. Я научился проводить аналогии между номерами строк и счетчиком команд, между операторами `GOTO` и `GOSUB` и командами `jr` и `call`.

Z80 также позволял мне делать ужасные в своем роде вещи: манипулировать порядком выполнения операторов. Например, я выработал привычку писать свой код так, что, если требовалось выполнить два шага, А и В, и шаг В сам по себе был полезной частью функциональности, я помещал А перед В, чтобы не приходилось вызывать В. Программа просто естественным образом переходила к шагу В после выполнения А. Такой стиль имел побочный эффект: было трудно сказать, как я попал в В — путем перехода из А или из какого-то другого места, но это не имело для меня значения, потому что я досконально знал свой код.

Нет, правда.

Еще я мог изменить стек, чтобы вернуться, минуя произвольное количество точек вызова, потратив всего одну инструкцию. Так было быстрее. Поскольку я досконально знал свой код, я мог таким способом ускорять свои игры. Все это имело значение.

Нет, правда имело.

Затем я пересел за ZX Spectrum: у него было больше оперативной памяти (16 Кбайт), а также цвет и звук! Однако с расширением масштабов моей платформы росли мои амбиции и мой код, и отлаживать его становилось все труднее. Иногда я не мог понять, откуда я попал в эту точку в программе и какой код уже был выполнен. Я быстро осознал, что слишком усложняю себе жизнь различными махинациями и манипуляциями, и стал искать компромиссы между скоростью выполнения и простотой понимания кода. Я решил, что дополнительные такты не стоят потери понимания кода, если я потом не смогу устранить все ошибки. Я узнал, что писать сверхбыстрый

¹ Zaks R. Programming the Z80. — Berkeley, CA: Sybex, 1979.

код Z80 весело и довольно легко, но отладить его практически невозможно: необходимо найти золотую середину между производительностью и удобочитаемостью. Это был ценный урок.

В результате я изменил стиль написания кода. Я разбивал его на многократно используемые части и детально описывал, где какая часть начинается и где заканчивается. Я перестал уменьшать указатель стека, переходить сразу к внутренним полезным частям больших функций. Жить стало значительно легче.

Далее я перешел на Atari ST с процессором Motorola 68000, работающим на частоте 8 МГц, и поистине выдающимся объемом ОЗУ 512 Мбайт. Мой стиль программирования, основанный на детальной организации всех частей программы, продолжал поддерживать меня. Каждая часть имела только одну точку входа, и управление всегда возвращалось туда, откуда был сделан вызов. Я говорил всем, что это единственный верный путь: я был ортодоксально фанатичен в своем рвении.

Как оказалось, я был не единственным, кто писал такой код. Программисты на FORTRAN и COBOL тоже приходили к подобным ограничениям, если обжигались на собственной неосторожности вольного обращения с текстами программ. Так родилась простая мудрость: «один вход, один выход». Всякая функция должна иметь одну и только одну точку входа, и должно быть только одно место, куда она возвращает управление: в точку вызова.

Правило один вход, один выход было частью философии структурного программирования, возникшей из письма Эдсгера Дейкстры (Edsger Dijkstra) редактору под заголовком *GOTO statement considered harmful* (о вреде оператора GOTO)¹. Книга *Structured Programming*² по-прежнему прекрасно читается, и я советую прочитать и письмо, и книгу. Они рассказывают, как развивалось программирование в течение десятилетия.

К сожалению, старые привычки трудно искоренить. Кроме того, причины появления этих привычек с течением времени стираются из памяти. Эволюция программирования, инновации уменьшают значимость старой мудрости. Функции в C++ сами избавляют от необходимости следовать идее «одного входа». Синтаксис C++ делает невозможным переход

¹ www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF

² *Dahl O. J., Dijkstra E. W., Hoare C. A. R. Structured Programming* (Дал У., Дейкстра Э., Хоор К. Структурное программирование).

в середину функции. Точно так же нет возможности вернуться куда-либо еще, кроме как в точку вызова (это не касается обработки исключений и сопрограмм).

Вы можете спросить, какое это имеет отношение к множественным операторам возврата. Дело в том, что, к сожалению, в умах программистов произошла путаница с предлогами, и «выход В одно место» превратился в «выход ИЗ одного места».

Роковая случайность.

ГАРАНТИЯ ОЧИСТКИ

Посмотрим на пример, демонстрирующий мудрость использования единственного оператора возврата. Предположим, что следующая функция вызывает старую библиотеку на C, экспортирующую функции для получения, управления и отображения ресурсов по целочисленному идентификатору:

```
int display(int ID)
{
    auto h = get_handle_to_resource(ID);
    if (h == 0) {
        return 0;
    }
    auto j = manipulate_resource(h);
    if (j < 0) {
        release_handle(h);
        return 0;
    }
    auto k = supply_resource_to_system(h, j);
    if (k < 0) {
        return 0; /* программист забыл освободить дескриптор */
    }
    display_resource(h);
    release_handle(h);
    return 1;
}
```

В этом фрагменте кода можно видеть весьма распространенную ситуацию: функция получает дескриптор ресурса и должна освободить его, прежде чем он выйдет из области видимости. Инженер забыл освободить дескриптор перед третьим оператором возврата. Одно из решений этой

проблемы — оставить в функции единственный оператор возврата, перед которым выполняется освобождение дескриптора, например, так:

```
int display(int ID)
{
    auto result = 0;
    auto h = get_handle_to_resource(ID);
    if (h != 0) {
        auto j = manipulate_resource(h);
        if (j >= 0) {
            auto k = supply_resource_to_system(h, j);
            if (k >= 0) {
                display_resource(h);
                result = 1;
            }
        }
    }
    release_handle(h);
    return result;
}
```

Ан нет, это неправильно. Функцию `release_handle` следует вызывать, только если дескриптор был успешно получен. Попробуем иначе:

```
int display(int ID)
{
    auto result = 0;
    auto h = get_handle_to_resource(ID);
    if (h != 0) {
        auto j = manipulate_resource(h);
        if (j >= 0) {
            auto k = supply_resource_to_system(h, j);
            if (k >= 0) {
                display_resource(h);
                result = 1;
            }
        }
    }
    release_handle(h);
}
return result;
```

Этот подход плохо масштабируется для длинных функций с большим количеством условных ветвей, потому что каждое условие требует дополнительного отступа, что снижает удобочитаемость кода.

Но функции в любом случае должны быть маленькими, то есть названный аргумент не самый веский. Такой подход также вводит дополнительное

состояние в виде отслеживания возвращаемого значения, что предполагает дополнительную нагрузку на читающего код. Нагрузка, казалось бы, небольшая, но есть риск ее увеличения, особенно после того, как будет вычислено правильное значение. И этот риск сам тоже будет расти с ростом кода функции.

Преимущество же этого подхода заключается в том, что функция `release_handle` вызывается независимо от происходящего, хотя она и должна быть вызвана в правильной ветви `if`.

Гарантия очистки дескриптора — весьма веский аргумент в пользу единственного оператора возврата. Это разумный совет.

Ошибка в первой реализации `display` заключалась в том, что дескриптор ресурса освобождался не перед каждым выходом из функции. Для исправления мы перестроили функцию так, что все соответствующие пути заканчивались одним вызовом `release_handle`, после которого можно безопасно вызвать `return`.

Выдающейся особенностью C++ является детерминированная, определяемая программистом очистка, обеспечиваемая деструкторами. И никто не сможет опровергнуть такое мнение. Введение этой возможности одним махом устранило целый класс ошибок. Под детерминированностью я подразумеваю, что вы точно знаете, когда будет вызван деструктор. В случае с автоматическими объектами, которые создаются в стеке, этот момент наступает, когда они покидают область видимости.

ИДИОМА RAI

Для гарантий выполнения кода вместо потока управления безопаснее использовать идиому, известную как получение ресурсов при инициализации (Resource Acquisition Is Initialization, RAII). Согласно этой идиоме функции получения и освобождения дескрипторов объединяются в одну структуру:

```
int display(int ID)
{
    struct resource {
        resource(int h_) : h(h_) {}
        ~resource() { release_handle(h); }
        operator int() { return h; }
    };
}
```

```

private:
    int h;
};

resource r(get_handle_to_resource(ID));
if (r == 0) {
    return 0;
}
auto j = manipulate_resource(r);
if (j < 0) {
    return 0;
}
auto k = supply_resource_to_system(r, j);
if (k < 0) {
    return 0;
}
display_resource(r);
return 1;
}

```

Обратите внимание, что этот код сигнализирует об ошибках не путем вызова исключений, а путем выполнения операторов возврата с разными значениями, сигнализирующими об успехе или неудаче. Если бы это была библиотека C++, а не библиотека C, можно было бы ожидать, что функция будет генерировать исключение, а не просто возвращать. Как в таком случае выглядел бы наш пример?

```

void display(int ID)
{
    struct resource {
        resource(int h_) : h(h_) {}
        ~resource() { release_handle(h); }
        operator int() { return h; }

private:
    int h;
};

resource r(get_handle_to_resource(ID));
auto j = manipulate_resource(r);
supply_resource_to_system(r, j);
display_resource(r);
}

```

Конечно, функции могли бы принимать пользовательские типы, а не `int`, но давайте оставим этот нюанс в стороне, это сейчас не главное, и пример создан для другой цели.

Теперь у нас вообще нет явного оператора возврата, что вполне ожидаемо. В конце концов, функция просто что-то делает, даже не сигнализируя об

успехе или неудаче. При использовании исключений для сигнализации об ошибках нет необходимости в операторе возврата: код предполагает успех и просто вызывает исключение, если терпит неудачу. Он не вычисляет значение и не возвращает его.

Эта структура очень полезна и стоит того, чтобы извлечь ее из этой функции и сделать доступной для других пользователей. В реальной жизни существует много программ, взаимодействующих с библиотеками на С и содержащих нечто подобное:

```
template <class T, class release_fn>
struct RAII
{
    RAII(T t_) : t(t_) {}
    ~RAII() { release_fn r; (t); }
    operator T() { return t; }

private:
    T t;
};
```

Здесь `T` обычно является встроенным или другим простым типом.

Следует признать, что исключения используются далеко не во всех программах на C++. Генерация исключения требует раскручивания стека и уничтожения всех автоматических объектов, созданных между `try` и `catch`. Для этого программа вынуждена хранить дополнительные данные, занимающие память. C++ используется в самых разных окружениях, иногда с весьма жесткими ограничениями на объем доступной памяти или время выполнения. Одному из авторов однажды довелось быть свидетелем реальной драки на стоянке из-за буфера 1 Кбайт, внезапно ставшего доступным после какой-то хитрой оптимизации.

Компиляторы позволяют запретить обработку исключений, что дает в результате меньшие по размеру и более быстрые двоичные файлы. Но это опасное решение.

Во-первых, вам придется довольствоваться неполноценной обработкой ошибок. И это в то время, как `dynamic_cast` генерирует исключение, если приведение к типу ссылки оказывается невозможным. Стандартная библиотека вызывает исключение, если ей не удается разместить новый объект в памяти. Неправильный доступ к объекту `std::variant` тоже вызовет исключение.

Во-вторых, уменьшение размеров двоичных файлов и увеличение скорости выполнения не гарантируются. Добавление сложного кода обработки

ошибок может свести на нет все преимущества и привести к дополнительным затратам. Однако если это важно, если это непреложное условие, то инженеры будут писать код, приспособливаясь к отсутствию поддержки исключений. Код получается не очень красивым, но ничего иного не остается, когда нет другого выхода.

Одному из авторов как-то довелось быть свидетелем реальной драки на стоянке из-за буфера 1 Кбайт, внезапно ставшего доступным после какой-то хитрой оптимизации.

Если ваше окружение поддерживает обработку исключений, то предпочтительнее использовать единственный оператор возврата, передающий успешно вычисленное значение в точку вызова. Наличие нескольких операторов возврата можно рассматривать как сигнал, что функция пытается сделать слишком многое.

ПИШИТЕ ХОРОШИЕ ФУНКЦИИ

В Core Guidelines имеется около 50 рекомендаций, касающихся функций. Мы рассматриваем две из них в отдельных главах, однако считаем нужным упомянуть некоторые другие в контексте использования нескольких операторов возврата. Например, рекомендация «F.2. Функция должна выполнять одну логическую операцию» предлагает разбивать большие функции на более мелкие и с подозрительностью относиться к функциям с большим количеством параметров. Один из побочных эффектов следования этому совету: ваши функции, скорее всего, будут иметь единственный оператор возврата с результатом функции.

То же относится к рекомендации «F.3. Функции должны быть короткими и простыми». В ней приводится пример функции, занимающей 27 строк и включающей три оператора `return`. Однако в последнем примере, где часть логики помещена в две вспомогательные функции, главная функция занимает почти в три раза меньше строк и по-прежнему содержит три оператора `return`, определяющих результат на основе входных параметров.

Рекомендация «F.8. Отдавайте предпочтение чистым функциям» — отличный, но трудный в реализации совет. Чистыми называют функции, не ссылающиеся на состояние за пределами своей области видимости. Это делает их распараллелимыми, простыми для понимания, проще поддающимися оптимизации и опять же более короткими и простыми.

Важно отметить, что железных советов очень и очень мало. Типичные рекомендации, как правило, реализуются на уровне языка. Например, совет «не допускайте утечки ресурсов» реализуется функциями-деструкторами и интеллектуальными указателями из стандартной библиотеки.

Наличие единственного оператора `return` может служить признаком соблюдения других хороших практик, но это не универсальное правило. Это дело вкуса и стиля. Взгляните на следующую функцию:

```
int categorize1(float f)
{
    int category = 0;
    if (f >= 0.0f && f < 0.1f) {
        category = 1;
    }
    else if (f >= 0.1f && f < 0.2f) {
        category = 2;
    }
    else if (f >= 0.2f && f < 0.3f) {
        category = 3;
    }
    else if (f >= 0.3f && f < 0.4f) {
        category = 4;
    }
    else {
        category = 5;
    }
    return category;
}
```

и сравните ее с этой функцией:

```
int categorize2(float f)
{
    if (f >= 0.0f && f < 0.1f) {
        return 1;
    }
    if (f >= 0.1f && f < 0.2f) {
        return 2;
    }
    if (f >= 0.2f && f < 0.3f) {
        return 3;
    }
    if (f >= 0.3f && f < 0.4f) {
        return 4;
    }
    return 5;
}
```

Какая из них лучше? Они обе делают одно и то же. Для обеих любой компилятор, скорее всего, сгенерирует идентичный код. Вторая содержит несколько операторов возврата, но сама короче. Первая содержит дополнительное состояние, но четко определяет последовательность взаимоисключающих условий, а не скрывает эту информацию за отступами. В зависимости от вашего опыта, используемых языков программирования, усвоенных советов по оформлению кода и целого ряда других условий вы будете выбирать тот или иной подход. Объективно ни один из этих подходов не лучше другого; и совет «Каждая функция должна иметь только один оператор возврата» нельзя назвать однозначно разумным.

ПОДВЕДЕМ ИТОГ

Все мы хотели бы иметь однозначные и ясные правила, которые можно было бы легко и точно применять, но таких золотых правил очень мало, и обычно они реализованы на уровне языка программирования. Правило единственного оператора возврата устарело и подлежит отмене в контексте C++.

- Разберитесь, из чего проистекает то или иное общераспространенное правило.
- Различайте возврат результатов и вызов исключений.
- Определите, какие правила являются делом вкуса.



НЕ НАВРЕДИТЕ СЕБЕ

Глава 2.1 Р.11. Инкапсулируйте беспорядочные конструкции, а не разбрасывайте их по всему коду.

Глава 2.2 И.23. Минимизируйте число параметров в функциях.

Глава 2.3 И.26. Если нужен кросс-компилируемый ABI, используйте подмножество в стиле C.

Глава 2.4 С.47. Определяйте и инициализируйте переменные-члены в порядке их объявления.

Глава 2.5 СР.3. Сведите к минимуму явное совместное использование записываемых данных.

Глава 2.6 Т.120. Используйте метапрограммирование шаблонов, только когда это действительно необходимо.

ГЛАВА 2.1

Р.11. Инкапсулируйте беспорядочные конструкции, а не разбрасывайте их по всему коду

В этой главе мы познакомимся с инкапсуляцией и тесно связанными с ней идеями сокрытия и абстрагирования информации. Эти три понятия часто используются взаимозаменяемо, и нередко за счет ясности. Прежде чем начать, подготовим почву для дальнейшего обсуждения, создав парсер (синтаксический анализатор), а затем вернемся к этим трем понятиям.

ВСЕ ОДНИМ ГЛОТКОМ

Одно из качеств, присущих великому инженеру, — способность замечать, когда что-то начинает выходить из-под контроля. Мы все отлично замечаем, когда что-то уже вышло из-под контроля. Кто из нас не начинал обзор кода с фразы: «Здесь намешано довольно много всего, поэтому было трудно разобраться». К сожалению, этот навык редко позволяет предвидеть начало возникновения беспорядка в коде.

Рассмотрим пример. Допустим, программе необходимо прочитать некоторые настройки из внешнего файла, указанного в командной строке. Они объявлены в виде пар «ключ/значение». Возможных параметров всего с десяток, но мы, как умные инженеры, решаем написать для этой цели отдельную функцию. Назовем ее `parse_options_file`. Она будет принимать имя файла, извлеченное из командной строки. Если имя файла не указано, то функция не будет вызвана. Учитывая сказанное, получаем сигнатуру функции:

```
void parse_options_file(const char*);
```

Функция имеет простую реализацию: открыть файл, читать настройки строку за строкой, изменять состояние соответствующим образом — и так до тех пор, пока не будет достигнут конец файла. Вот как это могло бы выглядеть в коде:

```
void parse_options_file(const char* filename)
{
    auto options = std::ifstream(filename);
    if (!options.good()) return;
    while (!options.eof())
    {
        auto key = std::string{};
        auto value = std::string{};
        options >> key >> value;
        if (key == "debug")
        {
            debug = (value == "true");
        }
        else if (key == "language")
        {
            language = value;
        }
        // и т.д. ...
    }
}
```

Круто! Можно легко добавлять новые и удалять старые настройки, и все в одном месте. Если файл содержит неподдерживаемые настройки, то можно просто уведомить пользователя в конечном объявлении `else`.

Через пару дней понадобилось добавить новую настройку, значением которой может быть несколько слов. Пустяки! Ведь есть возможность просто прочитать весь текст до конца строки. Для этого создадим буфер на 500 символов и сделаем из него строку. Это будет выглядеть так:

```
else if (key == "tokens")
{
    char buf[500];
    options.getline(&buf[0], 500);
    tokens = std::string{buf};
}
```

По-настоящему крутое программное решение! Придуман простой и гибкий способ анализа настроек. Теперь можно добавить код для обработки любой настройки! Отлично!

На следующей неделе коллега заявляет, что некоторые настройки действительны, только если пользователь установил флаг отладки. Нет проблем: устанавливаем флаг отладки в начале функции, а значит, сможем проверить

его позже и применять настройки, только когда это допустимо. Имейте в виду, сейчас мы отслеживаем состояние, и об этом обстоятельстве нужно помнить.

Спустя месяц файлы с недопустимыми настройками начинают вызывать волнение и клиенты просят применять настройки, только если все они действительные. Что ж, вздохнем и примемся за дело. Можно создать объект с настройками, содержащий новое состояние, и возвращать его, только если все встреченные настройки действительные. Здесь пригодится тип `std::optional`. Извлекаем свою функцию из репозитория и обнаруживаем, что бывшая прекрасная, аккуратная, ухоженная, элегантная, красивая, неповторимая функция пользуется, наверное, большой популярностью, потому что другие инженеры добавили в нее свои настройки. На текущий момент функция обрабатывает 115 настроек. Поддерживать ее будет проблематично, но ничего страшного, ведь это просто набор значений, которые будут получены в функции, а затем переданы в точку вызова...

Но не спешите. Остановитесь и подумайте. У вас есть 600-строчная функция, наполненная самыми разными данными и огромным количеством условных операторов. Вы действительно считаете себя новым Дональдом Кнутом?¹ Что здесь произошло?

Мы создали (или позволили создать) запутанную конструкцию. Одна функция с несколькими экранами в длину и несколькими вкладками в глубину, все еще неуклонно растущая, — вот что это такое. Эта функция пострадала от расплазания области видимости, и нужно обязательно найти способ ограничить ее, прежде чем все вокруг рухнет: мы-то прекрасно понимаем, что это только вопрос времени, когда ошибки начнут сыпаться как из рога изобилия и нам придется постоянно поддерживать и чинить это чудовище. Необходимо обязательно выпросить время для рефакторинга, прежде чем катастрофа поглотит и код, и вашу карьеру разработчика.

ЧТО ОЗНАЧАЕТ ИНКАПСУЛИРОВАТЬ ЗАПУТАННУЮ КОНСТРУКЦИЮ

В начале главы мы говорили, что рассмотрим понятия инкапсуляции, сокрытия информации и абстрагирования. Пришло время выполнить обещание.

Инкапсуляция — это процесс включения одного или нескольких элементов в единую сущность. Как ни странно, эта сущность так и называется:

¹ https://ru.wikipedia.org/wiki/Кнут,_Дональд_Эрвин

инкапсуляция. C++ предлагает несколько механизмов инкапсуляции. Наиболее очевидный из них — класс. Возьмите некоторые данные и функции, заключите их в пару фигурных скобок и поместите ключевое слово `class` (или `struct`) и идентификатор перед открывающей скобкой. Другой механизм — перечисления: возьмите множество констант, заключите их в пару фигурных скобок, а впереди поместите ключевое слово `enum` и идентификатор. Определения функций — еще одна форма инкапсуляции: возьмите набор инструкций, заключите их в пару фигурных скобок и поместите впереди идентификатор и пару круглых скобок, которые, в свою очередь, могут содержать параметры.

Но можно и продолжить! Не забывайте про пространства имен. Возьмите множество определений и объявлений, заключите их в пару фигурных скобок и поместите впереди ключевое слово `namespace` и необязательный идентификатор. Файлы с исходным кодом тоже являются инкапсуляцией: возьмите множество определений и объявлений, поместите их в файл и сохраните в файловой системе под некоторым именем. Модули — это первый новый механизм инкапсуляции, появившийся за долгое время. Они подобны файлам с исходным кодом: возьмите множество определений и объявлений, поместите их в файл, добавьте ключевое слово `export` в начале и сохраните в файловой системе под некоторым именем.

Любой имеющий опыт работы с модулями скажет, что инкапсуляция — это еще не все. В каждом из перечисленных примеров мы просто объединили какие-то элементы вместе и дали им общее имя. Будучи умными программистами, мы всегда стараемся объединять взаимосвязанные элементы. Сокрытие информации — более тонкая работа, требующая принятия более обдуманных решений. Собрав элементы в единую сущность, нужно решить, какие из них должны быть доступны внешнему миру, а какие — нет. Сокрытие информации подразумевает наличие некоторой инкапсуляции, но инкапсуляция не означает, что имеет место сокрытие информации.

Некоторые механизмы инкапсуляции в C++ поддерживают сокрытие информации. Классы предлагают уровни доступа. Приватные члены скрыты от клиентов структуры. Скрывая реализацию, мы освобождаем клиентов от бремени поддержки инвариантов класса и предотвращаем нарушение этих инвариантов. Перечисления не поддерживают сокрытия информации, то есть они не дают возможности сделать доступными только несколько членов. Функции прекрасно скрывают информацию: идентификатор и тип возвращаемого значения общедоступны, но реализация скрыта. Пространства имен могут открывать объявления и скрывать определения, охватывая

несколько файлов. Заголовочные файлы и файлы с исходным кодом по своим действиям во многом подобны модулям.

Давайте посмотрим, как в нашем примере может помочь инкапсуляция. Имеется множество настроек, которые обрабатываются в одной функции. Также у нас может быть отдельная функция для обработки каждой настройки. Эти функции вызываются в операторах `if`, проверяющих ключ. Функции могут возвращать логическое значение в зависимости от допустимости значения параметра.

Сокрытие информации подразумевает наличие некоторой инкапсуляции, но обратное неверно: инкапсуляция не означает, что имеет место сокрытие информации.

Выглядит неплохо: все настройки инкапсулированы в отдельные функции, и можно легко добавлять дополнительные функции для новых настроек, нужно лишь расширить функцию синтаксического анализа для каждого из них. Мы даже можем зафиксировать возвращаемые значения для проверки файла с настройками. Но нам все еще нужно создать объект, который будет применять настройки, если они допустимы. Поэтому его тоже нужно расширить при добавлении новой настройки. Об этом можно упомянуть в документации, и в любом случае другие инженеры получат хорошие примеры в виде уже реализованных функций.

Ваше чутье должно подсказать вам, где именно что-то может пойти не так. Вы все еще надеетесь, что другие инженеры поступят правильно, добавляя новые настройки. Но они ведь могут неправильно понять природу процесса проверки, забыть проверить функцию в операторе `if` или ошибиться в тексте с именем настройки. Да, вы значительно улучшили ситуацию, но иногда инкапсуляции и сокрытия информации недостаточно. Чтобы решить оставшиеся проблемы, придется использовать большую пушку: абстрагирование.

НАЗНАЧЕНИЕ ЯЗЫКА И ПРИРОДА АБСТРАКЦИИ

Абстрагирование — коварное слово. Дело не в том, что результатом абстрагирования является абстракция, так же как результатом инкапсуляции является инкапсуляция. Рассмотрим процесс абстрагирования, применив тот же подход, что мы использовали для знакомства с идеями инкапсуляции и сокрытия информации.

Буквально «абстрагирование» означает «вычленение». В контексте программирования это означает выявление и выделение важных частей проблемы, их вычленение и отбрасывание остального. Мы отделяем важные части от деталей реализации и помечаем абстракции идентификаторами. Возьмем для примера функции: мы объединяем набор инструкций в единую сущность и даем ей имя. Функция — это абстракция с именем, значимым для предметной области. То же относится к классам: класс — это абстракция с именем, значимым для предметной области, содержащая соответствующие функции, которые моделируют поведение, подразумеваемое именем.

Выбор того, что именно должно лежать в пределах абстракции, а что оставаться за ее границами, — целое искусство. Этим абстрагирование отличается от простой инкапсуляции. Кроме того, мы использовали слово «искусство», потому что нет механистического метода определения, где провести черту между тем, что относится к абстракции, а что — нет. Способность выделять абстракцию приходит с практикой и опытом.

Но вернемся к нашей задаче. Попытаемся проанализировать файл с парами «ключ/значение» и применить результаты к окружению, если они допустимы. Функция имеет хорошее говорящее имя: `parse_options_file`. Проблема заключается в безопасном добавлении произвольных пар «ключ/значение». Действительно ли идентификация полного набора пар имеет отношение к `parse_options_file`? Дело происходит внутри области видимости? Можно ли отделить настройки от функции?

В настоящий момент мы просто извлекаем ключи из файла и проверяем их в постоянно растущей цепочке операторов `if-else`, потому что оператор `switch-case` не поддерживает выбор вариантов по строкам. Такая конструкция выглядит как ассоциативный контейнер. На самом деле для этой задачи идеально подошло бы сопоставление ключей с указателями на функции с помощью встроенного класса `map`. В этом случае наша функция существенно упростилась бы и превратилась в одно обращение к `map` и последующий вызов соответствующей функции.

```
auto options_table = std::map<std::string, bool(*)(std::string const&) >
{{"debug"s, set_debug},
 {"language"s, set_language}}; // сюда добавляются другие настройки

void parse_options_file(const char* filename) {
    auto options = std::ifstream(filename);
    if (!options.good()) return;
    while (!options.eof()) {
        auto key = std::string{};
```

```
    auto value = std::string{};
    options >> key >> value;
    auto fn = options_table.find(key);
    if (fn != options_table.end()) {
        (*fn->second)(value);
    }
}
```

Важной частью этой функции является анализ файла с настройками и выполнение некоторых действий для каждого из ключей. К сожалению, вместе с этим утрачена способность значений содержать пробелы. Оператор шеврона (`>>`) прекращает извлечение, встретив пробел. Мы еще вернемся к этой проблеме.

И все же эта функция выглядит намного лучше. Осталось только инициализировать сопоставление ключей и указателей на функции. К сожалению, мы только что переместили проблему из одного места в другое. Инициализатор — это еще одно место, где пользователи могут споткнуться: легко забыть обновить инициализатор. Может быть, эту задачу можно автоматизировать?

Да, можно. Вместо сопоставления ключей с указателями на функции можно сопоставить их с объектами функций с помощью конструкторов и создавать статические объекты, а не функции. Конструктор может вставить адрес объекта в `map`. На самом деле все объекты-функции могут наследовать один базовый класс, который сделает это автоматически. Кроме того, имея базовый класс, мы сможем добавить проверочную функцию (`validate`) перед основной совершающей (`commit`). Кажется, все сходится.

```
auto options_table = std::map<std::string, command*>{};  
  
class command {  
public:  
    command(std::string const& id) {  
        options_table.emplace(id, this);  
    }  
    virtual bool validate(std::string const&) = 0;  
    virtual void commit(std::string const&) = 0;  
};  
  
class debug_cmd : public command {  
public:  
    debug_cmd() : command("debug"s) {}  
    bool validate(std::string const& s) override;  
    void commit(std::string const& s) override;  
};  
debug cmd debug cmd instance;
```

```
class language_cmd : public command {
public:
    language_cmd() : command("language"s) {}
    bool validate(std::string const& s) override;
    void commit(std::string const& s) override;
};
language_cmd language_cmd_instance;
```

Что дальше? Несмотря на то что мы анализируем файл с настройками, по сути, мы просто читаем последовательность символов. Эта последовательность не обязательно должна быть получена из файла: она может поступить из самой командной строки. Поэтому давайте переименуем функцию в `parse_options` и изменим входной параметр на `std::istream`. Если параметр, переданный в командной строке, не распознается как один из поддерживаемых ключей, его можно распознать как имя файла, попытаться открыть его и выполнить рекурсивный вызов.

```
void parse_options(std::istream& options) {
    while (options.good()) {
        auto key = std::string{};
        auto value = std::string{};
        options >> key >> value;
        auto fn = options_table.find(key);
        if (fn != options_table.end()) {
            if ((*fn->second)->validate(value)) {
                (*fn->second)->commit(value);
            }
        } else {
            auto file = std::ifstream(key);
            parse_options(file);
        }
    }
}
```

Теперь, имея отдельные объекты функций для всех ключей, мы не ограничены инициализацией данных и можем рассматривать каждый ключ как команду. Более того, получено простое средство для создания и выполнения скриптов (сценариев). Если при использовании первой версии, представленной в начале этой главы, инженер должен был расширять функцию, то теперь ему достаточно определить новый класс, наследующий класс `command`, и переопределить методы `validate` и `commit`.

Итак, мы ушли от одной огромной функции синтаксического анализа, заменив ее небольшой связанный функцией и словарем анализируемых объектов. Попутно удалось получить средство анализа параметров командной строки. Все это было достигнуто за счет внимательного изучения,

что и к какой части задачи относится. То, что сначала было запутанной конструкцией, превратилось в ясное и простое в сопровождении средство скриптования с небольшими дополнениями. В результате все стороны в выигрыше.

УРОВНИ АБСТРАКЦИИ

Другой способ перейти от одной функции синтаксического анализа к более мелким наборам функций — сгруппировать связанные действия в отдельные функции, например, так:

```
void parse_options_file(const char* filename)
{
    auto options = std::ifstream(filename);
    if (!options.good()) return;
    while (!options.eof())
    {
        auto key = std::string{};
        auto value = std::string{};
        options >> key >> value;
        parse_debug_options(key, value);
        parse_UI_options(key, value);
        parse_logging_options(key, value);
        // и т. д. ...
    }
}
```

Этот подход действительно решает проблему инкапсуляции запутанной конструкции: теперь у нас есть несколько функций, каждая из которых обрабатывает отдельную категорию. Однако это решение лишь сместило проблему, но не улучшило ситуацию. Инженеры, на плечи которых ляжет сопровождение, должны будут решить, в какую функцию добавить свой синтаксический анализатор. Когда эти функции станут слишком большими, придется решать вопрос об их дальнейшем разделении. Такой подход не учитывает уровни абстракции.

Чтобы понять суть уровней абстракции, рассмотрим семиуровневую модель OSI¹. Эта модель делит коммуникационную систему на уровни абстракции. Каждый уровень определяет интерфейс для следующего уровня, но не для предыдущего. Инженеры работают на том уровне, который соответствует их специальности. Например, гипотетически вы ин-

¹ https://ru.wikipedia.org/wiki/Сетевая_модель_OSI

женер-программист, а не инженер-электронщик. На уровне 1, физическом уровне, вы чувствовали бы себя очень некомфортно, и вам было бы гораздо удобнее работать на уровне 7, прикладном уровне. Возможно, вы слышали термин «full-stack-инженер». Такой инженер способен продуктивно работать на всех уровнях. Но это мифическое создание вряд ли встречается в реальности.

Уровни абстракции в задаче синтаксического анализа можно описать так, как показано ниже.

1. Уровень потоковой передачи, доставляющий поток данных в...
2. Уровень синтаксического анализа, доставляющий отдельные символы в...
3. Уровень словаря, сопоставляющий символы с токенами и доставляющий их в...
4. Уровень токенов, проверяющий ввод и обновляющий значения.

Все эти абстракции являются отдельными, непересекающимися частями задачи.

АБСТРАКЦИЯ ПУТЕМ РЕФАКТОРИНГА И ПРОВЕДЕНИЯ ЛИНИИ

Ключом к абстрагированию является осознание, где провести линию, разделяющую уровни. Как отмечалось выше, это искусство, а не наука, но есть три вещи, на которые следует обратить внимание.

Во-первых, это излишняя детализация. Оцените, тратится ли время на выполнение действий, которые кажутся совершенно не связанными с поставленной задачей? В Core Guidelines, в описании обсуждаемой рекомендации, приводится пример бесконечного цикла `for`, выполняющего чтение файла, проверку и перераспределение памяти для структуры. Можно также представить проектирование вычурных структур данных для локального использования. Можно ли использовать такие структуры данных вне этого контекста? Случится ли такое когда-нибудь? Если ответ положительный, то нужно взять эту структуру и переместить ее в общую библиотеку. Разделение деталей по разным библиотекам — это форма абстракции, которая применима как к примеру в Руководстве, так и к понятию извлечения структур данных.

Во-вторых, примите во внимание многоократное повторение. Какие закономерности имеют место? Приходилось ли копировать какие-то фрагменты кода? Можно ли выразить их в виде отдельной функции или шаблона? Выделите этот код в функцию, дайте ей имя и радуйтесь, что определили абстракцию.

В-третьих, держите в голове все то же «изобретение колеса». Этот аспект немного отличается от повторения и является комбинацией первых двух пунктов. На определение и именование фундаментальных алгоритмов было потрачено много времени. Убедитесь, что в стандартной библиотеке нет ничего похожего.

Повторение — это признак, что существует алгоритм, ожидающий, пока его откроют, а хороший алгоритм — это просто краткое описание того, что делает фрагмент кода. В 2013 году Шон Пэрент (Sean Parent) выступил с докладом под названием C++ Seasoning¹, большую часть первой половины которого он посвятил мантре «откажись от простых (необработанных) циклов». Он советовал использовать существующий алгоритм, такой как `std::find_if`, или реализовать известный алгоритм в виде шаблона функции и внести его в библиотеку с открытым исходным кодом, или разработать совершенно новый алгоритм, написать о нем статью и стать знаменитым, выступая с докладами. Это отличный совет, который поможет вам избавиться от запутанного кода с повторениями.

ПОДВЕДЕМ ИТОГ

Запутанный код не позволяет читателю с первого взгляда понять происходящее. Избегайте этого.

- Выявляйте шаблоны в существующем коде или алгоритмах, желательно сразу, как только они появляются; извлекайте и абстрагируйте в их собственные функции.
- Определяйте разные уровни абстракции.
- Определяйте, как инкапсулировать все эти части.

¹ <https://www.youtube.com/watch?v=W2tWOdgzXHA>

ГЛАВА 2.2

I.23. Минимизируйте число параметров в функциях

СКОЛЬКО ОНИ ДОЛЖНЫ ПОЛУЧАТЬ?

Взгляните на следующее объявление функции:

```
double salary(PayGrade grade, int location_id);
```

Приятно видеть double salary («удвоить зарплату») в начале строки кода, но, к сожалению, цель этой функции — не удвоить вам зарплату, а сообщить о размере зарплаты для заданного разряда в заданном месте. Кроме того, как отмечалось выше, для финансовых расчетов предпочтительнее использовать целочисленные типы, поэтому давайте изменим объявление этой функции так:

```
int salary(PayGrade grade, int location_id);
```

Подобное можно увидеть в государственных контрактах: заработка госслужащих часто привязана к определенному разряду, но также зависит от местоположения рабочего места. В Великобритании, например, некоторым госслужащим в Лондоне платят больше из-за более высокой стоимости жизни в столице.

Это совершенно нормальная, обычная функция. Не похоже, что она полагается на внешнюю информацию, — по всей вероятности, она просто обращается к базе данных для получения данных. Она выглядит как первая попытка написать функцию: простая, недвусмысленная и непрятязательная.

Затем в один прекрасный день парламент или какой-либо другой орган издает новый акт с изменившимися требованиями, согласно которым зарплата должна рассчитываться не только исходя из тарифа и географии места, но и с учетом стажа работы. Например, чтобы удержать опытных сотрудников, было решено применить небольшой повышающий коэффициент к зарплатам сотрудников, проработавших долгое время.

Это не проблема: мы легко можем добавить в функцию параметр, представляющий стаж работы. Стаж будет измеряться в годах, поэтому определим параметр типа `int`:

```
int salary(PayGrade grade, int location_id, int years_of_service);
```

Возможно, увидев пару целочисленных параметров, вы уже понимаете, что может возникнуть ошибка, если соответствующие аргументы перепутать местами. Вы мысленно ставите галочку, чтобы посмотреть, сколько разных местоположений существует, и преобразовать второй параметр в более специализированный тип, определив перечисление.

Проходит время, механизм оплаты снова меняется, и возникает необходимость добавить еще один параметр — численность команды. Кто-то выразил недовольство, что уровни оплаты труда руководителей не отражают дополнительное бремя управления командами, насчитывающими более десяти человек. Вместо того чтобы добавить новый разряд, что потребовало бы огромного количества бюрократических переговоров, было решено добавить новое правило вычисления размера оплаты труда руководителей, которое, как и в случае с выслугой лет, вводит небольшой повышающий коэффициент. Для реализации этого правила мы добавляем четвертый параметр, который передает количество подчиненных:

```
int salary(PayGrade grade, int location_id, int years_of_service,
           int reports_count);
```

Граница `reports_count` зависит от уровня оплаты. Для руководителей больших команд вступает в силу множитель оплаты выше определенного уровня. Однако информация о том, применяется ли множитель, важна для других функций. После обширного обсуждения о преимуществах возврата `std::pair<int, bool>` по сравнению с добавлением указателя на `bool` в список параметров команда, выступавшая за `std::pair`, проиграла, и теперь сигнатура функции выглядит так:

```
int salary(PayGrade grade, int location_id, int years_of_service,
           int reports_count, bool* large_team_modifier);
```

Оказывается, есть несколько десятков мест, границы которых довольно размыты, поэтому тип перечисления не подходит для второго параметра.

Сейчас эта функция сильно перегружена. Наличие в сигнатуре трех целочисленных параметров, идущих подряд, — это потенциальная ловушка для невнимательных. Если у руководителя с семилетним стажем в подчинении имеется пять человек, то перемена местами этих двух значений при обращении к функции, скорее всего, даст кажущееся достоверным, но ошибочное

значение. Такую ошибку трудно обнаружить из-за свойственной человеку «слепоты к словам».

Кроме того, функция не только запрашивает информацию из базы данных, но также выполняет некоторые дополнительные вычисления, одно из которых зависит от другого. Функция называется `salary` (зарплата) и кажется безобидной, но в действительности за кулисами она выполняет множество действий. Если для приложения, использующего эту функцию, важна высокая производительность, то это может иметь значение.

УПРОЩЕНИЕ ЧЕРЕЗ АБСТРАГИРОВАНИЕ

Рассматриваемая рекомендация предполагает, что двумя наиболее распространенными причинами слишком большого количества параметров в функциях являются следующие.

1. Отсутствие абстракций.
2. Нарушение принципа «одна функция, одна ответственность».

Рассмотрим их подробнее.

Цель функции `salary` — вычислить значение для заданного состояния. Сначала функция имела два элемента состояния, но по мере изменения требований их количество выросло. Однако одно оставалось неизменным: функция вычисляла зарплату по информации о сотруднике. Поразмыслив после появления третьего параметра в сигнатуре функции, программисты решили, что разумнее инкапсулировать параметры в единую абстракцию и назвать ее `SalaryDetails`.

В этом и заключается отсутствие абстракции. Если у вас появился набор состояний, необходимый для достижения цели, и состояния эти, возможно, связаны между собой, то высока вероятность, что вы обнаружили абстракцию. Соберите эти состояния в один класс, дайте ему имя и сформируйте взаимные связи между ними в виде инвариантов класса.

Применив этот процесс к функции `salary`, получаем структуру `SalaryDetails`:

```
struct SalaryDetails
{
    SalaryDetails(PayGrade grade_, int location_id_, int years_of_service_,
                  int reports_count_);
    PayGrade pay_grade;
    int location_id;
```

```
    int years_of_service;
    int reports_count;
};
```

и сигнатуру функции:

```
int salary(SalaryDetails const&);
```

Это лишь часть необходимых улучшений. В сигнатуре конструктора все еще присутствует три целочисленных параметра, готовых, как капканы, к поимке невнимательных. На самом деле в Core Guideline имеется рекомендация, предупреждающая такую практику: «I.24. Избегайте смежных параметров одного типа из-за риска перепутать местами аргументы с разными значениями в вызове функции». Хорошо, что существуют методы смягчения этой проблемы, такие как строгая типизация, так что не все потеряно.

По мере изменения требований к вычислению заработной платы в структуру `SalaryDetails` могут вноситься соответствующие уточнения. Более того, `salary` можно сделать функцией-членом абстракции `SalaryDetails`, а `large_team_modifier` превратить в предикат, то есть в функцию, возвращающую `true` или `false`, и создать класс:

```
class SalaryDetails
{
public:
    SalaryDetails(PayGrade grade_, int location_id_, int years_of_service_,
                  int reports_count_);
    int salary() const;
    bool large_team_manager() const;

private:
    PayGrade pay_grade;
    int location_id;
    int years_of_service;
    int reports_count;
};
```

Клиентский код в таком случае мог бы выглядеть так:

```
auto salary_details = SalaryDetails(PayGrade::SeniorManager, 55, 12, 17);
auto salary = salary_details.salary();
auto large_team_manager = salary_details.large_team_manager();
```

Если решение с использованием функций-членов вам не подходит, то переменные-члены можно объявить общедоступными, а клиентский код будет выглядеть так:

```
auto salary_details = SalaryDetails(PayGrade::SeniorManager, 55, 12, 17);
auto salary = calculate_salary(salary_details, &large_team_manager);
```

Обобщим то, что обсуждали выше. Нам потребовалась функциональность для получения значения из некоторого состояния. Количество состоянийросло. Мы абстрагировали состояние в класс и добавили функции-члены, делающие то, что изначально требовалось от оригинальной функции.

Уделим немного времени и обсудим, откуда взялись данные, используемые в примере вызова функции. Мы явно указали 55, 12 и 17, но такой подход к передаче аргументов маловероятен. Скорее всего, где-то в системе существует класс `Employee`, содержащий эту информацию, и она просто передавалась в функцию `salary`, например, так:

```
for (auto const& emp : employees)
    auto final_salary = calculate_salary(
        PayGrade::SeniorManager, emp.location, emp.service, emp.reports);
```

Когда мы видим такой вызов функции, то сразу удивляемся, почему эта функция не член класса источника данных. Правомерен вопрос: «Почему `salary` не является функцией-членом класса `Employee`?»

Возможно, автор не может изменить класс `Employee`, так как он определен в стороннем коде. В таком случае лучше передать весь класс `Employee` в функцию `salary` через константную ссылку и позволить функции самой обращаться к классу, например:

```
for (auto const& emp : employees)
    auto final_salary = calculate_salary(PayGrade::SeniorManager, emp);
```

Оба решения уменьшают количество параметров, которые принимает функция `salary`, что и является целью данной рекомендации.

ДЕЛАЙТЕ ТАК МАЛО, КАК ВОЗМОЖНО, НО НЕ МЕНЬШЕ

Означает ли это, что наборы параметров всегда должны преобразовываться в классы?

Конечно, нет. Рекомендация лишь предлагает минимизировать количество параметров функций. В рамках x64 ABI существует соглашение по умолчанию о быстром вызове с четырьмя регистрами. Функция с четырьмя параметрами будет выполняться немного быстрее, чем функция, принимающая класс по ссылке. Вам решать, стоит ли заменять четыре параметра простых типов одним параметром-классом. Конечно, если функция принимает дюжину параметров, то определенно предпочтительнее создать

класс для инкапсуляции состояния. Но это не жесткое правило, а просто рекомендация, которую следует примерять к своему контексту.

Вторая часть обсуждения рекомендации посвящена нарушению правила «одна функция — одна ответственность». Согласно этому простому правилу, функция должна делать что-то одно, но качественно. Примером нарушения этого принципа может служить функция `realloc`.

Если вы будете действовать как добропорядочный программист на C++, то можете никогда не столкнуться с этим зверем. Он живет в стандартной библиотеке C, где объявлен в заголовке `<stdlib.h>` с такой сигнатурой:

```
void* realloc(void* p, size_t new_size);
```

Эта функция решает несколько задач. Основная — изменение объема блока памяти, на который указывает `p`. Если говорить точнее, она увеличивает или уменьшает объем блока памяти, на который указывает `p`, до размера `new_size` в байтах. Если объем существующего блока нельзя просто увеличить, то функция выделяет новый блок требуемого размера, а затем копирует в него содержимое старого блока. Эта операция копирования не поддерживает семантику конструктора копирования, а просто копирует байты, поэтому порядочный программист на C++ вряд ли с ней столкнется.

Если в параметре `new_size` передать ноль, то поведение функции определится конкретной реализацией. Вы могли бы подумать, что в этом случае она просто освободит блок, но это верно не для всех реализаций.

Функция возвращает адрес нового или расширенного блока памяти. Если запрошен блок большего размера и в системе недостаточно памяти для его размещения, то вызывающему коду возвращается пустой указатель.

Итак, функция решает две задачи: изменяет размер блока памяти и, возможно, перемещает его содержимое. Явно видно смешивание разных уровней абстракции, и второе действие обусловлено несостоятельностью первого. Этую функциональность логично реализовать как единственную функцию с именем `resize`, например:

```
bool resize(void* p, size_t new_size);
```

Если у вас появился набор состояний, необходимый для достижения цели, и состояния эти, возможно, связаны между собой, то высока вероятность, что вы обнаружили абстракцию. Соберите эти состояния в один класс, дайте ему имя и сформируйте взаимные связи между ними в виде инвариантов класса.

Она могла бы просто попытаться увеличить или уменьшить размер блока, а в случае неудачи выделить новый блок и переместить в него данные. Отпала бы необходимость в поведении, определяемом реализацией. То есть хорошим тоном было бы разделить уровни абстракции, связанные с перераспределением памяти и перемещением данных.

Принцип «одна функция — одна ответственность» обусловлен связностью. В программной инженерии под связностью понимается степень взаимозависимости сущностей. Мы надеемся, что пример выше достаточно ясно демонстрирует высокую связность, существующую внутри функции. Высокая связность — это хорошо. Она способствует улучшению удобочитаемости и возможности повторного использования, а также снижению сложности. Если у вас не получается подобрать хорошее имя для своей функции, это может говорить о том, что она решает слишком много разных задач. Для чего-то сложного всегда нелегко подобрать хорошее имя.

В случае с классами высокая связность подразумевает тесную связь функций-членов и данных. Связность увеличивается, когда функции-члены выполняют небольшое количество связанных действий с небольшим набором данных. Высокая связность и слабая сопряженность часто идут рука об руку. Более полную информацию по этой теме можно получить в книге *Structured Design*¹, изданной более 40 лет тому назад, но продолжающей оставаться актуальной и по сей день.

ПРИМЕРЫ ИЗ РЕАЛЬНОЙ ЖИЗНИ

Если присмотреться внимательно, можно увидеть, что второй принцип просто иначе выражает первый. Когда функция накапливает обязанности, ее следует абстрагировать, превратить в очень конкретную запись в словаре предметной области. Давайте рассмотрим примеры, показанные в самом Core Guideline.

Первый — функция `merge`. Это чудище имеет следующую сигнатуру:

```
template <class InIt1, class InIt2, class OutIt, class Compare>
constexpr OutIt merge(InIt1 first1, InIt1 last1,
                      InIt2 first2, InIt2 last2,
                      OutIt dest, Compare comp);
```

¹ Yourdon E., Constantine L. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design (2 ed.). — N. Y.: Yourdon Press, 1978 (*Йордан Э. Структурное проектирование и конструирование программ*).

Сигнатура выглядит более или менее читабельной, но даже притом, что она скопирована с cppreference.com, для этой книги нам пришлось ее немного сократить, чтобы уместить по ширине страницы, напечатать несколько раз, внимательно вычитать и исправить несколько ошибок.

Здесь явно есть возможность абстрагирования. Функция принимает набор итераторов, отмечающих пару диапазонов. Начиная с C++20, можно явно определять диапазоны в виде пар итераторов, отмечающих начало и конец диапазонов. Это позволяет упростить функцию, объединив вместе первые четыре параметра с итераторами:

```
template <class InRng1, class InRng2, class OutIt, class Compare>
constexpr OutIt merge(InRng r1, InRng r2, OutIt dest, Compare comp);
```

Детали реализации диапазонов находятся на более низком уровне абстракции. В данном же случае нас интересует только объединение двух диапазонов. Другой способ определить диапазон — передать указатель на его начало и количество элементов. Во втором примере в Руководстве приводится функция с такой сигнатурой:

```
void f(int* some_ints, int some_ints_length);
```

Еще один объект, появившийся в C++20, — `std::span`. Core Guidelines сопровождается библиотекой поддержки, которая так и называется — Guidelines Support Library. Это набор классов, определенных в пространстве имен `gs1`, которые можно использовать для следования некоторым рекомендациям. Класс `std::span` был разработан на основе `gs1::span` и реализован в точности так, как описано выше: указатель на начало последовательности данных и длина объединены в один объект, что дает следующую сигнатуру:

```
void f(std::span<int> some_ints);
```

В обоих примерах мы выполнили абстрагирование на основе параметров.

Существует еще один способ определения диапазона — с указателем и контрольным значением. Этот метод встроен в язык в виде строкового литерала и выражается в виде указателя на массив символов с завершающим нулем (NUL, 0x00). Этот заключительный нулевой символ и есть контрольное значение. Более специализированную версию объекта `std::span` — `std::string_view` — можно создать из пары итераторов, из указателя и счетчика, а также из указателя и контрольного значения.

ПОДВЕДЕМ ИТОГ

Абстракции могут конструироваться множеством способов, но часто наступает момент, когда клиент хочет большего от своей функции. Мы в ответ не можем просто сказать: «Нет, я не собираюсь добавлять еще один параметр, нарушая принцип I.23» — и сложить руки на груди, выражая протест. Это не очень любезно по отношению к вашим коллегам, которые одновременно являются вашими клиентами. Но что делать, если нет больше значимых абстракций, которые можно было бы ввести?

Большое количество параметров может служить сигналом, что сложность выходит из-под контроля и нужно что-то делать. Если параметры не поддаются дальнейшему абстрагированию, то, возможно, проблема заключается в самом идентификаторе функции. Если к функции предъявляется слишком много требований, то это говорит о ее важности для предметной области. Однако вместо попытки заключить эту важность в одну абстракцию, возможно, настала пора подумать о разделении функции. Можно, например, написать перегруженные версии для разных сценариев использования или разделить действия на несколько функций с разными именами, соответствующими конкретным операциям. Можно также написать шаблонную функцию. Всегда есть другие варианты, кроме добавления дополнительных параметров.

Важно помнить, что, независимо от причины добавления дополнительных параметров в функцию, к этому нельзя относиться легкомысленно и этот шаг следует рассматривать как крайнюю меру и повод для переосмыслиния природы самой функции. Минимизируйте количество параметров, разделяйте похожие параметры, упрощайте функции и радуйтесь обнаружению абстракций.

- Большое количество параметров увеличивает нагрузку на пользователя.
- Организуйте параметры в структуры; возможно, это поможет обнаружить скрытые абстракции.
- Рассматривайте наличие нескольких параметров как признак, что функция решает слишком много задач.

ГЛАВА 2.3

I.26. Если нужен кросс-компилируемый ABI, используйте подмножество в стиле C

СОЗДАВАЙТЕ БИБЛИОТЕКИ

Создавать библиотеки на C++ просто. Скомпилируйте файлы с исходным кодом, объедините их в библиотеку, опишите экспортируемые элементы в заголовочном файле или в модуле, если ваш компилятор поддерживает их, и передайте файл библиотеки с заголовочным файлом или определением модуля вашему клиенту.

К сожалению, это еще не все. Существует множество мелких деталей, которые необходимо учитывать. Цель этой рекомендации — показать, как свести к минимуму накладные расходы, не допустить, чтобы огромный объем работы лег на ваши плечи в какой-то непредсказуемый момент в будущем, и реализовать таким образом возможность создавать библиотеки, которые будут служить сообществу много лет.

Вспомним, как работает компоновщик. Он сопоставляет отсутствующие символы из объектных файлов или библиотек с экспортами символами из других объектных файлов или библиотек. Объявления в заголовке ссылаются на определения в файле библиотеки.

Например, представьте, что вы написали библиотеку с функцией:

```
id retrieve_customer_id_from_name(std::string const& name);
```

и заголовок, объявляющий ее. Так можно создать пакет функциональности с заголовочным файлом, чтобы вашему пользователю не приходилось

тратить время на перекомпиляцию. Возможно, ваш исходный код является вашей собственностью, и вы не хотите передавать его другим.

Судя по имени, эта функция извлекает буфер с символами по полученной ссылке на строку, посыпает эту строку в запросе к базе данных, находящейся где-то в облаке, и возвращает идентификатор, извлеченный из ответа на запрос. Реализация строки может содержать длину, за которой следует указатель на буфер, поэтому получение буфера с символами и передача его в базу данных является тривиальной задачей.

А теперь представьте, что ваша библиотека стала безумно популярной, возможно, потому, что в базе данных полно полезной информации и каждый хочет получить кусочек. Число ваших клиентов сначала исчислялось сотнями, потом тысячами и, наконец, десятками тысяч, и все они с радостью отдают вам свои деньги в обмен на разработанную вами функциональность. Внезапно начинают появляться жалобы, что библиотека дает сбой во время выполнения именно этой функции. Струйка жалоб превращается в поток, а затем в настоящий ревущий водопад гнева. Вы приступаете к исследованиям и замечаете, что обновился набор инструментов: изменились компилятор, компоновщик и стандартная библиотека.

Следующим шагом вы выполняете сборку библиотеки и модульных тестов с новым набором инструментов и запускаете полное тестирование. Все тесты выполняются успешно. Нет никакого сбоя. Вы пытаетесь воспроизвести ошибку, но безрезультатно. Так что же произошло в действительности?

Проблема заключалась в изменении определения `std::string`. Когда вы только создали библиотеку, тип `std::string` был реализован как длина, за которой следовал указатель. Но потом ваши клиенты перешли на использование обновленного набора инструментов и стандартной библиотеки, реализующей тип `std::string` как указатель, за которым следует длина. Теперь ваша функция получает `std::string const&` с иной организацией памяти. При попытке разыменовать указатель происходит разыменование длины и попадание в защищенную область памяти, приводящее к завершению по ошибке.

Повторная сборка библиотеки и модульных тестов выполнялась с помощью нового набора инструментов, поэтому повсюду использовалось определение `std::string`, в котором первым следует указатель, а за ним длина. Как результат, все тесты выполнились успешно. Ошибка проявлялась, только когда новый код клиента вызывал старый код вашей библиотеки.

ЧТО ТАКОЕ ABI

Произошедшее обусловлено изменением в ABI. Прикладной программный интерфейс (API) вашей библиотеки не изменился, но изменился ABI. Термин ABI может быть не знаком вам. Эта аббревиатура расшифровывается как Application Binary Interface — прикладной двоичный интерфейс. Прикладной программный интерфейс (API) является руководством для людей и определяет действия, которые можно выполнять с библиотекой, а прикладной двоичный интерфейс (ABI) является руководством для машин и определяет особенности взаимодействия с библиотекой. ABI можно интерпретировать как скомпилированную версию API.

ABI определяет не только особенности размещения объектов в стандартной библиотеке, но и порядок передачи их функциям. Например, согласно System V AMD64 ABI¹ первые шесть целочисленных аргументов или аргументов-указателей передаются функциям в регистрах RDI, RSI, RDX, RCX, R8 и R9. Этому ABI следуют Unix и Unix-подобные операционные системы. Также к ведению ABI относятся обработка и распространение исключений, формат пролога и эпилога функции, организация таблицы виртуальных методов, соглашения о вызове виртуальных функций и т. д.

Необходимость следования требованиям ABI — одна из причин, почему код, скомпилированный для одной операционной системы, не будет работать в другой, даже на одной и той же аппаратной платформе, такой как x86. У этих операционных систем могут быть разные ABI. Конечно, если бы существовал один общий ABI, то это не имело бы значения, но ABI тесно связаны с характеристиками оборудования. Привязка к единственному ABI приведет к снижению производительности. Библиотека, которая не использует типы из другой библиотеки в сигнатурах своих функций, не может пострадать от нарушения ABI, если та библиотека изменится. Со временем все библиотеки меняются, и от этого никуда не деться.

Поэтому важно поддерживать стабильный ABI. Любое изменение функции, типов возвращаемых значений, количества и порядка аргументов или спецификации `noexcept` — все это нарушения ABI и изменения API. Изменение определений типов или структур данных в приватном интерфейсе не является изменением API, но *является* нарушением ABI.

¹ https://wiki.osdev.org/System_V_ABI

Даже простое декорирование имен (name mangling) может нарушить ABI, потому что ABI может определять стандартный способ уникальной идентификации функций для поддержки связывания вместе библиотек, написанных на разных языках, например C и Pascal. Возможно, вам уже приходилось видеть объявление `extern "C"` в заголовочных файлах. Это сигнал для компилятора, что объявленные имена функций должны добавляться в таблицу экспортации с применением схемы именования, используемой компилятором C на этой платформе.

Предыдущая гипотетическая проблема с библиотекой, которая принимает `std::string`, вряд ли вас побеспокоит. Современные компоновщики просто не позволяют клиенту скомпоновать библиотеки с разными ABI. Для этого может использоваться, например, встраивание версии ABI в декорированные имена символов или в объектный файл. Тогда при попытке связать конфликтующие версии ABI компилятор может выдать сообщение об ошибке. Трудозатраты, связанные с ABI, заключаются в выявлении таких конфликтов и являются одной из обязанностей инженера.

Запомните правило: «Если нужен кросс-компилируемый ABI, используйте подмножество в стиле C».

Таким образом, мы объяснили природу кросс-компилируемого ABI, обосновали важность его соблюдения. Теперь перейдем к подмножеству в стиле C.

СОКРАЩАЙТЕ ДО АБСОЛЮТНОГО МИНИМУМА

Итак, что включает подмножество в стиле C? Типы языка C часто называют встроенными. Это элементарные типы, в определении которых не участвуют никакие другие типы. Их можно было бы назвать атомарными, но, к сожалению, в C++ этот термин имеет другое значение. Однако вы можете рассматривать их как основные строительные блоки всех других типов. К ним относятся¹:

```
void  
bool  
char  
int  
float  
double
```

¹ <https://ru.cppreference.com/w/cpp/language/types>

Некоторые из этих типов можно модифицировать, добавляя и убирая поддержку знака или изменения размер. Ключевые слова `signed` и `unsigned` можно применить к типам `char` и `int`. Ключевое слово `short` можно применить к `int`. Ключевое слово `long` можно применить к `int` и `double`. Более того, ключевое слово `long` можно дважды применить к `int`. Фактически ключевое слово `int` можно полностью опустить и оставить только модификаторы знака и/или размера. Модификаторы также можно применять в любом порядке, например, тип `long unsigned long` полностью допустим.

На первый взгляд это может показаться несколько громоздким. Если бы код, содержащий `long unsigned long`, был предоставлен на экспертную оценку, велика вероятность, что у экспертов возникли бы вопросы касательно обоснованности его использования. Наличие модификатора размера порождает каверзный вопрос: «Насколько велико число типа `int`?» Правильный ответ на него: «Зависит от реализации». Стандарт же, со своей стороны, определяет некоторые гарантии размеров, уменьшая неопределенность:

- `short int` и `int` имеют размер не менее 16 бит;
- `long int` имеет размер не менее 32 бит;
- `long long int` имеет размер не менее 64 бит;
- `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`.

Но конкретные размеры все-таки определяются реализацией. Этот набор вариантов называют моделью данных. Есть четыре модели, снискавшие широкое признание:

- LP32 или 2/4/4 (`int` – 16 бит, `long` и указатели – 32 бита) использовалась в Win16 API;
- ILP32 или 4/4/4 (`int`, `long` и указатели – 32 бита) использовалась в Win32 API и в некоторых системах Unix;
- LLP64 или 4/4/8 (`int` и `long` – 32 бита, указатели – 64 бита) используется в Win64 API;
- LP64 или 4/8/8 (`int` – 32 бита, `long` и указатели – 64 бита) используется в некоторых системах Unix.

Это подчеркивает значение слова «кросс-компилируемый» в рекомендации. Ваша библиотека должна работать и сегодня, и через год, и с вашим компилятором, и с компиляторами ваших клиентов. Если компиляторы используют тип `int` или указатели разных размеров, то они будут вести себя по-разному

и в вашей программе, а вы, возможно, этого и не предполагали.

Типы с плавающей точкой проще целочисленных, потому что их формат определяется другими стандартами, в частности стандартом ISO/IEC/IEEE 60559:2011 или, что то же самое, стандартом IEEE 754-2008.

Это обеспечивает их полную переносимость между платформами. Тип `char` дополняется спецификаторами `signed` и `unsigned`; все три типа — `char`, `signed char` и `unsigned char` — являются разными. Тип `unsigned char` обычно используется для работы с необработанной памятью (на уровне байтов).

Библиотека, которая не использует типы из другой библиотеки в сигнатурах своих функций, не может пострадать от нарушения ABI, если та библиотека изменится. Со временем все библиотеки меняются, и от этого никуда не деться.

Из вышесказанного явно следует, что изменение модели данных приводит к нарушению ABI. Впервые я¹ столкнулся с нарушением ABI (хотя и не понимал этого) в начале 1990-х, когда впервые установил Win32 на свою машину, где прежде стояла Windows 3.11. Внезапно все мои значения типа `int` стали занимать в два раза больше памяти, и некоторые мои программы перестали работать. Совсем недавно различия между моделями LLP64 и LP64 (изменился только размер `long`) причинили мне массу неудобств при разработке для Windows и MacOS, потому что первая использовала LLP64, а вторая — LP64.

Все, можете расслабиться и выдохнуть. Рекомендация звучит так: «Если нужен кросс-компилируемый ABI, используйте подмножество в стиле C». И тем не менее мы только что видели, насколько хрупкими являются встроенные типы. Но эта хрупкость проявляется только при перемещении между моделями данных. Кроме того, есть способ преодолеть опасности, связанные со сменой модели данных, — использовать целочисленные типы фиксированного размера, такие как `int32_t` и `uint8_t`. Они определены в C в заголовке `<stdint.h>` и в C++ в заголовке `<cstdint>`.

Чтобы соответствовать этой рекомендации, объявление функции, представленное в начале этой главы, не должно было содержать типа `std::string`, если предполагалось сохранить работоспособность при смене ABI, потому что его определение может измениться. Именно так случилось в GCC с внедрением поддержки C++11 и повлекло множество неприятностей. Функция должна была бы принимать `char*` и `int` или еще лучше `uint8_t`, потому что размеры этих типов не изменятся никогда.

¹ Короткий экскурс в историю с Гаем Дэвидсоном.

РАСПРОСТРАНЕНИЕ ИСКЛЮЧЕНИЙ

Похоже, проблема решается просто: нужно лишь передавать и возвращать встроенные типы с фиксированными размерами — и проблем не будет. Однако есть еще один способ передачи данных между функциями — через механизм обработки исключений. Когда возникает исключение, компилятор выделяет в памяти место для него, а затем начинает раскручивать стек, вызывая специальную функцию из стандартной библиотеки. На некоторых платформах эта функция перебирает все операторы `catch` с использованием данных, вставленных компилятором и соответствующих каждой функции. Ее цель — отыскать оператор `catch`, который обработает сгенерированное исключение.

Возможны два исхода: функция либо найдет, либо не найдет подходящий оператор `catch`. В последнем случае стандартная библиотека вызовет функцию `std::terminate`, которая, в свою очередь, вызовет `std::abort`. Она вернет управление системе, не проводя очистку ресурсов. Если функции, раскручивающей стек, посчастливится найти соответствующий оператор `catch`, он вернется вниз по стеку и освободит ресурсы, занятые каждой функцией, вызвав деструкторы и восстановив стек, после чего возобновит выполнение тела найденного оператора `catch`. Затем, по завершении оператора `catch`, память, выделенная для исключения, освобождается.

Оставим в стороне вопрос о том, что происходит, когда исключение возникает из-за нехватки памяти. Поведение системы в таком случае зависит от конкретной реализации. Механизм, выделяющий место в памяти для исключения, и формат данных, вставляемых компилятором после каждого тела функции, тоже зависят от реализации, как и средства, с помощью которых каждый оператор `catch` описывается в функции раскручивания стека. Вся эта информация записывается в двоичный файл во время компиляции, и в программе существуют две библиотеки, собранные с помощью двух разных версий компиляторов. То есть существует риск столкнуться с несовместимостью их ABI. Однако риск этот весьма невелик: формат данных с информацией для обработки исключений определяется ABI платформы, а типы исключений — библиотекой, и поставщики делают все возможное, чтобы не менять их. Однако если вы чувствуете неуверенность, то единственный вариант — гарантировать, что исключения не покинут вашу библиотеку, и снабдить каждую функцию в API спецификатором `noexcept` или `noexcept(true)`, в зависимости от вашего стиля программирования.

Как видите, чтобы гарантировать кросс-компилируемый ABI библиотеки, требуется приложить некоторые усилия. Это проявление опасностей

двоичной зависимости. Вы можете принять решение распространять библиотеку в виде исходного кода или скомпилированного двоичного файла. В нашем примере использовалось понятие проприетарного кода, а также упоминалась компиляция на стороне клиента. Это серьезный вопрос: к примеру, настройка и сборка библиотеки поддержки графического интерфейса, которую авторы используют в своей студии, занимает большую часть дня. Но доступны также двоичные версии библиотеки — нужно лишь определить версию набора инструментов, которая будет использоваться для компиляции и сборки, и щелкнуть на соответствующей кнопке загрузки на веб-сайте авторов.

Разработчики предоставляют множество библиотек, содержащихся в структуре каталогов. Именно эта структура отражает, определен ли флаг `NDEBUG` и связаны ли двоичные файлы статически или динамически. Этот набор из четырех перестановок легко может превратиться в набор из 12 перестановок, потому что на момент написания этих строк авторами было внесено предложение о поддержке трех разных конфигураций контрактов. Выбор с учетом определения `NDEBUG` статической или динамической компоновки и конфигурации контракта может оказаться непростой задачей.

Сказанное не относится к библиотекам, распространяемым в виде исходного кода. Если ваша библиотека состоит исключительно из шаблонов классов и функций, то ее не получится распространять в виде предварительно скомпилированного двоичного файла: для создания экземпляров шаблонных классов и функций необходим полный исходный код. Если же ответственность за сборку делегируется вашему клиенту, то все хлопоты, связанные с совместимостью ABI, также делегируются ему. Это верно только для той ситуации, когда ваша библиотека сама не зависит от предварительно скомпилированной двоичной библиотеки.

ПОДВЕДЕМ ИТОГ

Вы можете решить, что ваша библиотека не будет поддерживать требования к стабильности ABI. Возможно, у вас в руках лишь краткосрочная ее версия, пока не появится более комплексное и долговечное решение.

Пожалуйста, никогда, никогда не позволяйте появляться таким мыслям в вашей голове.

В 1958 году Министерство обороны США запустило автоматизированную систему управления контрактами с названием «Механизация служб

администрирования контрактов» (Mechanization of Contract Administration Services, MOCAS). Она была написана на COBOL и продолжала использоваться даже в 2015 году, получив в свое время обновление с зеленым экраном, заменившим ввод с перфокарты, а затем и веб-интерфейс. Хорошие вещи создаются на века и часто используются. Триллионы долларов прошли через MOCAS¹.

Язык C существует уже 50 лет, а C++ — 40. Повсеместное распространение C привело к заимствованию и использованию интерфейсов C в других языках, таких как Perl, Python, Lua, PHP, Ruby, Tcl и многих других. Интерфейс C превратился в общепринятый язык межплатформенных взаимодействий.

Существует немалое количество библиотек и программ с «древними» родословными. Например, библиотека libcurl, которая очень часто используется членами C++ сообщества, впервые была выпущена в 1997 году. Она по-прежнему занимает важное место в инфраструктуре игровой индустрии.

Разработчики Windows могут быть знакомы с библиотекой MFC (Microsoft Foundation Classes) — набором классов для разработки с использованием Windows SDK. Версия 1 этой библиотеки вышла в 1992 году и до сих пор широко используется.

Любой, кто пишет программное обеспечение для обработки изображений, скорее всего, будет использовать библиотеку libjpeg. Первая ее версия вышла в октябре 1991 года.

Библиотека iostreams была написана в 1985 году. Несмотря на то что в стандарте C++20 ее частично заменила библиотека fmt, первая из них все еще находит широкое применение.

Кажется маловероятным, что авторы этих библиотек всерьез думали, что их творения будут использоваться 20, 30 или 60 лет спустя. Но мы просим вас разрабатывать свои продукты с мыслью о будущем и устраниять все возможные препятствия для их использования в этом, пока гипотетическом, будущем времени. Следуйте этой рекомендации, и ваши библиотеки смогут присоединиться к разработкам старейшин экосистемы C++. Вам будет присвоен статус легенды, и ваша слава не померкнет. Избегайте проблем с ABI и создавайте свои ABI на основе подмножества в стиле C.

¹ <https://www.technologyreview.com/2015/08/06/166822/what-is-the-oldest-computer-program-still-in-use>

ГЛАВА 2.4

С.47. Определяйте и инициализируйте переменные-члены в порядке их объявления

Переменные-члены инициализируются строго в порядке объявления, вы не можете изменить это. Есть очень веские причины, почему это правило является непреложным, и мы опишем их в этой главе.

Но сначала рассмотрим пример. Брину, новому инженеру в крупной фирме, предложили написать сервис для определения количества людей в здании. Сервер, скрытый в недрах здания, обрабатывает события входа и выхода людей из здания. Он старый и медленный. На лицевой панели красными буквами выгравировано название забытой компании, выпускавшей оборудование и давно проданной ненасытному стартапу, обладающему неплохими финансовыми возможностями. Запросы к серверу довольно громоздки, и на их обработку уходит много времени. Новые правила безопасности требуют, чтобы число людей в здании было доступно по первому требованию немедленно, но, так как для оформления входа/выхода на стойке регистрации нужно около минуты, считается допустимым кэшировать значение, полученное последним запросом, и хранить его в течение 30 секунд. Это позволяет предотвратить накопление запросов, которые не получается обработать быстро. Вам, наверное, смешно, но это реальная история, только имена изменены.

Брин реализует следующее простое решение:

```
class population_service {
public:
    population_service(std::string query);
    float population() const;
```

```
private:
    mutable float current_population;
    mutable std::time_t expiry;
    std::string query;
};
```

Для обработки запроса на сервере создается экземпляр класса. Вызов `population()` проверяет истечение срока действия значения `population` и при необходимости обновляет его, используя запрос. Это типичный случай использования спецификатора `mutable`.

Брин пишет конструктор:

```
population_service::population_service(std::string query_)
{
    query = std::move(query_);
    expiry = std::chrono::system_clock::to_time_t(
        std::chrono::system_clock::now());
    current_population = population();
}
```

Функция `population` реализована просто. Все тесты успешно выполняются, и Брин отправляет код на проверку. Его просят перенести инициализацию из тела конструктора в список инициализации. Стремясь произвести хорошее впечатление, Брин быстро вносит исправления и выдает следующий код:

```
population_service::population_service(std::string query_)
: query(std::move(query_))
, expiry(std::chrono::system_clock::to_time_t(
    std::chrono::system_clock::now()))
, current_population(population())
{}
```

Брин отправляет его, и спустя некоторое время к нему приходит лидер проекта Бет.

- Брин, — спрашивает она, — ты проверил этот код?
- Ага, все тесты зеленые, — невинно отвечает он.
- Ты проверил его *после* того, как перенесли все в список инициализации? — упорствует Бет.
- Ну, в этом же нет необходимости? — говорит он. — Я же просто переместил код за пределы фигурных скобок.

Бет чуть не лишается дара речи и в уме начинает готовить лекцию на тему «Автоматизируйте тесты и всегда выполняйте их перед отправкой». Ей-то не нужно запускать код, чтобы увидеть проблему.

К сожалению, простое перемещение кода вверх меняет его поведение. Первое, что делает обновленный код, — инициализирует `current_population` вызовом `population()`. Оставим в стороне сомнительность вызова функции-члена в списке инициализаторов, но подобное выполнение кода явно не входило в первоначальные намерения инженера. Он-то хотел сначала инициализировать запрос `query` и время истечения срока действия `expiry`. Если не инициализировать эти значения правильно, то маловероятно, что вызов `population()` увенчается успехом.

Отметим для большей ясности, что переменные-члены инициализируются в порядке объявления. Это единственный возможный порядок инициализации. Никакие другие варианты не возможны и не доступны. В нашем случае переменная-член `current_population` объявлена первой, поэтому она первой инициализируется вызовом функции-члена `population()`. Следующей инициализируется переменная-член `expiry`, но тип `std::time_t` не имеет конструктора и обычно реализуется как встроенный целочисленный тип, поэтому он не будет инициализирован детерминированным значением. Наконец, `query` инициализируется конструктором перемещения.

Есть два возможных выхода: инициализировать `current_population` нулем, `expiry` — вызовом `now()`, а затем переместить запрос в `query`:

```
population_service::population_service(std::string query_)
    : current_population(0)
    , expiry(std::chrono::system_clock::to_time_t(
        std::chrono::system_clock::now()))
    , query(std::move(query_))
{}
```

Дополнительно можно вызвать `population()` в теле конструктора. Но, учитывая, что это дорогостоящая операция, требующая кэширования, мы не будем вызывать ее, пока значение действительно не потребуется.

Другое решение — переупорядочить переменные-члены в определении класса, например:

```
class population_service {
public:
    population_service(std::string query_);
    float population() const;
```

```
private:
    std::string query;
    mutable std::time_t expiry;
    mutable float current_population;
};
```

В этом случае функцию `population()` можно безопасно вызвать в списке инициализации, хотя на самом деле так не надо поступать по той же причине.

Это приводит нас к интересному отступлению, касающемуся особенностей размещения объектов в памяти. Рассмотрим следующую программу:

```
#include <iostream>
struct int_two_bools {
    bool m_b1;
    int m_i;
    bool m_b2;
};
int main() {
    std::cout << sizeof(int_two_bools);
}
```

Как вы думаете, какое число она выведет?

Мы собрали эту программу, используя x64 MSVC 19.28 в Compiler Explorer, запустили и получили ответ 12. Каждый элемент структуры занимает четыре байта (рис. 2.1).

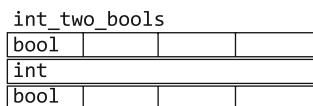


Рис. 2.1. Размещение в памяти целого числа и двух значений `bool`

Затем мы поменяли местами вторую и третью переменные-члены (рис. 2.2):

```
struct int_two_bools {
    bool m_b1;
    bool m_b2;
    int m_i;
};
```

Какое число теперь выведет программа? Вы можете проверить ее в Compiler Explorer и убедиться, что она выводит число 8.

| int_two_bools | | | |
|---------------|------|--|--|
| bool | bool | | |
| int | | | |

Рис. 2.2. Оптимизированное размещение в памяти целого числа и двух значений bool

Более того, если добавить еще одну переменную-член типа `bool` после `m_b2` (рис. 2.3):

```
struct int_two_bools {
    bool m_b1;
    bool m_b2;
    bool m_b3;
    int m_i;
};
```

вы получите *то же* число 8.

| int_two_bools | | | |
|---------------|------|------|--|
| bool | bool | bool | |
| int | | | |

Рис. 2.3. Оптимизированное размещение в памяти целого числа и трех значений bool

Для кого-то это может стать неожиданностью. В стандарте есть несколько слов о выравнивании, которые можно найти на странице [basic.align]¹. Суть сводится к тому, что отдельные объекты должны выравниваться по границам, соответствующим их размеру. В этом примере и для этой реализации `sizeof(bool)` дает значение 1, а `sizeof(int)` — значение 4. В первом определении `int_two_bools` первый член типа `bool` занимает один байт. Затем добавляются три дополняющих байта, чтобы член типа `int` начинался на границе с адресом, кратным четырем. Во втором определении два члена типа `bool` объявлены последовательно и должны выравниваться только по однобайтной границе, то есть перед членом типа `int` требуется добавить только два байта дополнения. В третьем определении требуется только один байт дополнения, потому что все три члена типа `bool` умещаются в четырех байтах.

Как видите, размер экземпляра класса может зависеть от порядка объявления членов. Это особенно актуально, если объем памяти ограничен

¹ <https://eel.is/c++draft/basic.align>

и программа должна создавать миллионы объектов: дополнительные четыре байта могут оказаться весьма ценными. Решение проблемы с порядком инициализации простым изменением порядка объявления членов может увеличить размер объекта.

Кто-то из вас может спросить: «Почему порядок имеет значение? Почему он должен соответствовать порядку объявления? Почему нельзя изменить порядок инициализации?» Кстати, некоторые компиляторы предупредят вас, если вы перечислите инициализаторы в неправильном порядке. Ответ заключается в гарантиях со стороны C++ уничтожения членов в порядке, обратном созданию.

Если бы порядок инициализации допускалось указывать в конструкторе, это сильно затруднило бы работу компилятора. При отсутствии явной реализации деструктора класса компилятору пришлось бы определить его в той же единице трансляции, где находится конструктор, и на основе последнего выяснить порядок уничтожения. При явном определении деструктора вам пришлось бы добавить что-то вроде списка деинициализации, выполняемого в конце тела деструктора. Кроме того, от вас потребовалось бы соблюсти синхронизацию между конструктором и деструктором. Такое решение открывало бы широкие возможности для ошибок.

«Нет, нет, все в порядке, я готов», — отвечаете вы с улыбкой. Вас вполне может устраивать такое положение вещей, но имейте в виду, что это не единственная проблема. Язык C++ позволяет перегружать конструктор. Вы уже понимаете, что порядок инициализации каким-то образом влияет на производительность, как и порядок членов. Но сможете ли вы гарантировать, что каждый конструктор будет оставаться синхронизированным?

«Да, да, конечно!» — отвечаете вы. Похоже, что вы уверены в своих силах. Однако иногда разные конструкторы могут иметь разные зависимости. В таких случаях невозможно гарантировать синхронизацию конструкторов.

«Ха! У нас может быть несколько деструкторов! Для каждого конструктора будет определяться соответствующий деструктор, чтобы каждая пара могла создавать и уничтожать члены в правильном порядке. Это великолепно! Напишу-ка я предложение в комитет!»

Ваше упорство достойно восхищения, но это не сработает. Представьте, что объект уничтожается в другой единице трансляции, отличной от той, где он создается. В таком случае компилятор не будет знать, какой деструктор

вызвать. Есть просто объект, покидающий область видимости, и нет никакой информации о конструкторе.

«Не проблема, мы просто добавим в класс тег, описывающий, как был создан объект, проверим его на этапе уничтожения и вызовем правильный деструктор».

Вы уже начинаете придумывать на ходу. Такое решение тоже имеет проблему: в каждый класс добавляется дополнительное поле для поддержки того, что редко используется. Вы увеличили накладные расходы времени выполнения для каждого класса. Создали абстракцию, за которую должны платить всегда, используется она или нет. Подобный акт противоречит одному из фундаментальных принципов языка: не платить за то, что не используется. В мире существуют миллиарды и миллиарды строк кода на C++. Если внедрить эту функцию в язык, негативному воздействию подвергнутся миллионы проектов. Этого нельзя допустить. Не будет нескольких деструкторов, и не будет возможности переопределения порядка инициализации. Члены инициализируются в том порядке, в котором они объявлены, и ни в каком другом. Мы надеемся, что вы поняли.

Выдающейся особенностью C++, на мой взгляд, является строго определенный порядок уничтожения объектов. Существует множество языков, автоматически управляющих временем жизни объектов. Но в C++ только *вы* решаете, когда объект должен прекратить существование и освободить занятые им ресурсы. Когда имя объекта с автоматическим классом хранения покидает область видимости, вызывается его деструктор и деструкторы его членов. Когда вы явно уничтожаете объект, размещенный в динамической памяти, вызываются его деструктор и деструкторы его членов. Когда программа завершается, уничтожаются все статические объекты в порядке, обратном их созданию: вызываются их деструкторы и деструкторы их членов. При уничтожении потока выполнения уничтожаются все локальные экземпляры в порядке, обратном их созданию: вызываются их деструкторы и деструкторы их членов.

Вам может показаться, что мы слишком усложняем, но это чрезвычайно ценное свойство языка, и если вы уже сталкивались с автоматическим управлением сроком жизни объектов, то поймете почему. Например, представьте, что у вас есть соединение с базой данных, хранящееся в виде члена в объекте, и такие соединения являются ограниченным ресурсом. Вы можете положиться на автоматическое закрытие соединения при вызове деструктора. Его не следует закрывать до уничтожения объекта — это надо делать,

только когда станет ясно, что оно больше не нужно. Именно для этого и нужен деструктор — функция, вызываемая в конце жизни объекта. Он гарантированно будет вызываться при выходе объекта из области видимости, чего сборка мусора просто не может предложить. Опираясь на область видимости, можно минимизировать время жизни объекта и максимизировать эффективность использования ресурсов. При программировании на языке, в котором время жизни объектов управляется какой-то другой системой, например средой времени выполнения, есть риск задержать освобождение соединения с базой данных до момента, когда среда выполнения произведет сборку мусора. Это может стать причиной больших разочарований, и тогда единственное решение — закрыть соединение вручную, когда вы будете уверены, что объект готов к уничтожению. Такой вид ручного управления временем жизни был характерен для C, и именно от него C++ стремится избавиться.

Строго определенный порядок уничтожения невозможен без одного ограничения, которое определяет эта рекомендация: все объекты должны создаваться в порядке, который можно определить статически во время компиляции. Для этого порядок создания выводится из определения класса и из порядка объявления переменных-членов. Существуют и другие способы определения порядка: обратный порядку объявления, алфавитный порядок идентификаторов элементов, использование специального члена, определяющего порядок числовыми тегами... мы уверены, что вы можете придумать множество других порядков. Однако все они кажутся несколько извращенными по сравнению с существующим четким порядком — порядком объявления.

В Core Guideline имеется родственная рекомендация: «C.41. Конструктор должен создавать полностью инициализированный объект». Один из способов обеспечить соответствие этой рекомендации — написать конструктор, создающий все члены в своем списке инициализации. Просьба лидера проекта, адресованная Брину, переместить инициализацию членов в список инициализации ясно свидетельствует о признании этого удобного подхода. Приведя список инициализации в соответствие с объявлениями членов, вы гарантируете, что каждый из них будет инициализирован до того, как начнется выполнение тела конструктора, где может быть произведена любая дополнительная работа, связанная с объектами вне класса, например создание журнала или подключение обработчиков обратных вызовов.

Беря на вооружение этот подход, помните о роли внутриклассовых инициализаторов членов. Они задают начальные значения по умолчанию

для объектов, поэтому конструктор по умолчанию становится ненужным. Если какие-либо переменные-члены имеют такие значения, убедитесь, что правильно назначаете их.

Здесь следует упомянуть так называемый «стиль с ведущими знаками препинания». Если посмотреть на код выше, можно заметить, что список инициализации для каждого конструктора был написан так, что инициализация каждого члена определялась в отдельной строке и перед каждым элементом списка стояли запятая или двоеточие (в случае инициализации первого члена). В числе других способов написания конструкторов можно упомянуть размещение всего списка инициализации в одной строке или использование более естественного подхода, когда запятые и двоеточие ставятся в конце каждой строки. Преимущество стиля с ведущими знаками препинания заключается в простоте изменения порядка членов: достаточно просто переместить строку целиком вверх или вниз без риска забыть убрать запятую в конце списка. Это мелочь, но она помогает избавиться от надоедливой ошибки компиляции.

Может возникнуть вопрос о накладных расходах: если член инициализируется в списке инициализаторов, а затем изменяется в теле конструктора, не приведет ли это к созданию избыточного кода? К счастью, у нас есть так называемое правило «как если бы». Вот как оно описано в стандарте на странице [intro.abstract]¹: «...соответствующие реализации необходимы для эмуляции (и только) наблюдаемого поведения абстрактной машины...» Это позволяет компилятору преобразовывать код с целью оптимизации. В примере выше член `current_population` может быть инициализирован нулем в списке инициализации, а затем изменен вызовом функции `population()` в теле конструктора, а компилятор, заметив вызов `population()`, может определить избыточность первой инициализации.

Не бойтесь добавлять явно избыточный код в список инициализации. Компилятор удалит его, если сможет. Цель добавления кода — сообщить компилятору, что делать, и предоставить ему как можно больше информации и контекста. Чем больше информации вы дадите компилятору, тем больше у него будет шансов генерировать оптимальный код. Конструктор должен создать полностью инициализированный объект, готовый к работе, не имеющий ненастроенных членов. Если конструктор не может этого сделать, значит, где-то в вашей абстракции кроется ошибка, которую нужно исправить как можно скорее. Это предупреждение, на которое следует

¹ <https://eel.is/c++draft/intro.abstract>

откликнуться немедленно, перепроектировав класс или сузив абстракцию, чтобы класс можно было инициализировать полностью.

Есть некоторые, так скажем, трюки, которые можно использовать для выполнения какой-то экзотической инициализации в списке инициализации. Напомним, что список инициализации формируется из серии выражений. В этом примере `current_population` инициализируется числом 0, `expiry_time` — вызовом `std::chrono::system_clock::to_time_t(std::chrono::system_clock::now())`, а `query` — перемещением параметра конструктора. В списке инициализации не допускается выполнять произвольный код, и можно вызывать лишь существующие функции. Разрешается использовать и тернарный оператор, так как он является выражением.

Для соблюдения этих требований раньше было принято объявлять в классах приватные статические функции для вспомогательного использования в выражениях инициализации. Это было разочаровывающее зрелище: интерфейс, заполненный функциями, которые можно было использовать только один раз, что увеличивало умственную нагрузку и бремя обслуживания.

С появлением в C++11 лямбда-выражений ситуация изменилась. Вместе с ними возникла идиома под названием IILE (Immediately Invoked Lambda Expression — «немедленно вызываемое лямбда-выражение»):

```
example::example(int a, int b, int c)
: x([&](){
    ...тело функции...
}())
{}
```

Конструктор класса `example` принимает три параметра и инициализирует `x`, объявляя лямбда-выражение и немедленно выполняя его за счет добавления пары круглых скобок. Этот способ удобно использовать для инициализации константных объектов, когда требуется выполнить обширные вычисления, которые невозможно последовательно применить к константному значению.

Однако этот прием просто перемещает проблему из одного места в другое: приватные статические функции теперь превратились в лямбда-выражения инициализации. Это лишь попытка использовать новый блестящий молоток для решения старой проблемы очистки апельсина, тогда как правильное решение по-прежнему состоит в том, чтобы пересмотреть абстракцию. Если на этапе инициализации требуется обширное манипулирование состоянием, то, возможно, где-то в данных все еще прячется невскрытая абстракция.

В обзоре рекомендации «I.23. Минимизируйте число параметров в функциях» мы обсудили некоторые подходы к уменьшению количества параметров функции, один из них заключался в формировании структуры для передачи конструктору. Не игнорируйте такого рода подсказки при разработке кода. Поиск абстракций — лучшее, что можно сделать при создании программного обеспечения.

ПОДВЕДЕМ ИТОГ

- Тело конструктора менее ограничено, чем список инициализации.
- Инициализация членов зависит от порядка их объявления. Это обусловлено необходимостью поддержки определенного порядка их уничтожения.
- Устраняйте несоответствия, изменяя порядок объявления членов, если это не приводит к увеличению потребления памяти из-за выравнивания.
- Избегайте решения проблем перебрасыванием кода: относитесь к проблемам зависимостей как к подсказкам, указывающим на существование пока не вскрытой абстракции.

ГЛАВА 2.5

СР.3. Сведите к минимуму явное совместное использование записываемых данных

ТРАДИЦИОННАЯ МОДЕЛЬ ВЫПОЛНЕНИЯ

Раздел Core Guidelines, касающийся конкуренции и параллелизма, посвящен обстоятельству, хорошо известному разработчикам на C++: писать многопоточные приложения намного сложнее, чем однопоточные. Этот вид программирования вводит уникальный класс ошибок, сбивающих с толку неосторожных и невнимательных. Отлаживать несколько потоков выполнения чрезвычайно сложно, потому что основной способ отладки, пошаговое выполнение кода, — это строго последовательный процесс.

Давайте познакомимся с предысторией вопроса. Почти наверняка вам приходилось видеть диаграммы, подобные изображенной на рис. 2.4.

Процессор последовательно выполняет инструкции программы. Одни инструкции загружают данные из первичного хранилища, другие манипулируют данными, а третьи сохраняют результаты в первичное хранилище. Некоторые инструкции управляют порядком, в каком процессор будет выполнять следующие инструкции.

Это отличный способ обучения основам программирования. Извлечение данных, обработка, сохранение и управление потоком выполнения — все, что нужно для написания программ.

Однако, когда в игру вступает реальное оборудование, картина немного меняется. Например, разные типы хранилищ имеют разные характеристики производительности. Вы уже не можете полагаться на то, что для извлечения

данных из произвольных мест потребуется одинаковое количество времени. Внедрение кэш-памяти в процессоры для настольных ПК в 1990-х годах выявило значительные потери времени, когда процессор простояивает, ожидая завершения операций загрузки и сохранения данных. Еще больше времени теряется, когда в операции чтения/записи вовлекаются внешние хранилища, такие как диски или твердотельные накопители. И еще больше, когда операции чтения и записи выполняются с удаленными хранилищами по сетевым кабелям или Wi-Fi.



Рис. 2.4. Блок-схема компьютера

Часто программам в процессе выполнения требуется извлекать какие-то данные, хранящиеся вне основной памяти на удаленном устройстве. И в операционной системе существуют функции, вызывая которые можно открывать соединения с этими устройствами. Далее мы будем рассматривать файл на локальном диске.

Программа должна создать буфер для приема данных в память, а затем предложить операционной системе открыть файл на локальном диске, прочитать данные в буфер и закрыть файл. Было бы идеально, если бы программа могла оставить операционной системе выполнение операций

с устройством, а сама заняться своим делом и получить уведомление, когда данные будут прочитаны с диска в буфер. И действительно, такая возможность имеется во всех современных операционных системах. Существует два потока выполнения: один создает буфер, а другой заполняет его.

Этот тип конкурентного выполнения относительно прост. Если правильно обрабатывать ошибки ввода/вывода, не пытаться читать буфер раньше времени и периодически проверять завершение операции с устройством, то никаких проблем не возникнет. Преждевременное чтение буфера — это класс ошибок, известный как гонка за данными. Гонка за данными возникает при следующих условиях:

- два или более потока выполнения одновременно обращаются к одной и той же ячейке памяти;
- один из потоков выполняет запись в память;
- потоки никак не управляют доступом к этой памяти.

К сожалению, чем сложнее программа, тем проще попасть в состояние гонки за данными. Допустим, ваш код не проверяет признак готовности буфера. Возможно, новичок, присоединившийся к вашему проекту, не понимает, как осуществляется ввод/вывод, и полагает, что программа блокируется вызовом функции чтения из файла и данные уже будут находиться в буфере, когда она вернет управление.

Ошибка здесь заключается в том, что незаполненный буфер доступен двум потокам и это позволяет потоку, создавшему буфер, попытаться прочитать данные до того, как они будут готовы. Одно из решений этой проблемы — скрыть буфер до момента, когда он заполнится: поручите потоку ввода/вывода создать и заполнить буфер, а затем вернуть его вызывающему потоку после заполнения.

ПОДОЖДИТЕ, ЭТО ЕЩЕ НЕ ВСЕ

В начале 1990-х, только закончив университет¹, я стал разработчиком Windows и работал на компьютерах с Windows. Я написал много мультимедийного программного обеспечения: очень хотел писать игры, но занимался тем, что умел. И это было нескучно.

¹ Экскурс в историю с Гаем Дэвидсоном. Дополнение к сказанному.

Одна из особенностей жизни разработчиков того времени — внезапное, неожиданное зависание системы. Плохо написанная программа приводила к остановке сеанса работы, и если у пользователя не получалось вызвать на экран диспетчер задач, чтобы уничтожить зависший процесс, то единственным выходом оставалось нажать кнопку сброса. К сожалению, в среде, в которой я работал, редко удавалось вызвать диспетчер задач при зависании. Чтобы обезопасить себя от потерь, я быстро научился время от времени сохранять промежуточные плоды своего труда, но это довольно сильно напрягало.

Я писал презентации для производителей рабочих станций, оказывая помощь их отделам продаж, и однажды на моем столе появилась рабочая станция Windows NT 3.51 с двумя процессорами Intel Pentium. У меня уже был опыт работы с серверами с несколькими процессорами, но не с подобными настольными компьютерами. Впечатления от перехода на новую технику превзошли мои ожидания. Через день, после установки технического окружения, я был готов к работе, и первое, что я заметил, — насколько чаще у меня получалось вызвать диспетчер задач и уничтожить зависший процесс. Теперь я мог спокойно делать свою работу и наслаждаться процессом разработки.

Конечно, будучи любознательным разработчиком, я первым делом проверил возможность использования обоих процессоров в одной программе. Я решил написать декодер, который распаковывал бы резервные копии из сети и записывал их на локальный диск. Один процессор читал и писал, а другой — распаковывал.

Менее чем через час моя программа зависла. И причина была мне совершенно непонятна. Я был очень внимателен, ожидая заполнения буферов перед распаковкой: у каждого из них была своя блокировка, которая сбрасывалась при заполнении буфера. Я потратил уйму времени, пытаясь понять, в чем причина проблемы. Как ни странно, когда код выполнялся в отладчике в пошаговом режиме, он работал нормально. Более того, когда я запускал код на других машинах, он тоже работал нормально.

Я не буду подробно описывать безумие, пережитое мною в течение следующих 12 дней (и ночей), пока я пытался понять, что, черт возьми, происходит. Оказалось, что проблема заключалась в синхронизации: каждый поток ждал, когда будет снята блокировка с буфера, заблокированного другим потоком. Это явление было уникальным и характерным именно для многопроцессорных машин. Я не мог воспроизвести его больше нигде в здании.

Так я столкнулся со своей первой взаимоблокировкой.

Взаимоблокировка возникает, когда каждый из двух потоков выполнения ждет освобождения ресурса, удерживаемого другим. Итак, каждый поток ждал освобождения буфера, заблокированного другим потоком, прежде чем завершить работу и снять блокировку со своего буфера. На однопроцессорных машинах все происходило так, как и ожидалось: один поток заполнял буфер и снимал свою блокировку, позволяя другому потоку обработать его. В системе с двумя процессорами оказалось возможным заблокировать оба потока, каждый из которых ждал завершения другого.

Как только я понял суть происходящего, я пересмотрел логику и исправил ошибку. Однако это был еще не конец. Другой код, который я написал раньше, тоже не работал на этой машине. Ситуации, очень редкие на однопроцессорных компьютерах, неожиданно часто проявлялись на этой многопроцессорной машине.

Мне стало ясно, что целью многопроцессорных машин было выполнение нескольких процессов и превращение Windows в более удобную платформу. Запуск нескольких потоков в одной программе явно не стоил приложенных усилий. Это было интересное академическое упражнение, но я крепко обжегся и не планировал возвращаться к этой конкретной идиоме.

Одной из особенностей разработки в 1990-х годах был неуклонный рост скорости процессоров. Мой первый рабочий компьютер был оснащен процессором Intel 80286, работающим на частоте 8 МГц. Это был 1992 год, и для того времени компьютер был несколько слабоват, но все еще широко использовался. К концу десятилетия появился Pentium III, работающий на частоте до 1,13 ГГц. Я быстро привык к тому, что процессоры становятся все быстрее. Я даже обнаружил, что для этого явления есть название: закон Мура, согласно которому количество транзисторов на кристалле удваивается каждые два года¹.

Конечно, это не закон в полном смысле, а всего лишь наблюдение. К сожалению, все хорошее когда-нибудь заканчивается, и в конце концов физика взяла верх. В декабре 2004 года Герб Саттер, один из авторов Core Guidelines, написал статью под названием *The Free Lunch Is Over*².

¹ <https://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html>

² www.gotw.ca/publications/concurrency-ddj.htm (перевод на русский язык: <https://habr.com/ru/post/145432/>. – Примеч. пер.).

Он осветил проблему с законом Мура и предсказал жесткую необходимость поворота программного обеспечения в сторону организации конкурентного выполнения для максимального использования потенциала будущих процессоров с растущим числом ядер. Он заметил отсутствие понимания конкуренции среди большинства программистов и особо подчеркнул, что эффективность и оптимизация приобретут особую важность как никогда ранее.

Впихивание все большего количества транзисторов в процессор уже не увеличивало скорость вычислений. Примерно в конце 2006 года я столкнулся со своим первым двухъядерным процессором. Да, процессоры уже не могут работать быстрее, но они могут за то же время делать больше, что тоже хорошо, верно?

Конечно, это не так. Два ядра способны выполнять ваш код в два раза быстрее, но только если им не приходится ждать друг друга. Я немного упал духом. Но я знал, что это произойдет, и не был разочарован. Условия гонки и взаимоблокировки стали частью жизни разработчика, пытавшегося выжать из процессора все больше и больше. Я привык к потокам ввода/вывода: теперь мир заполнялся вычислительными потоками. Многопоточные программы — новая реальность, особенно в моей области игр. Обыденным делом стали процессоры с дюжинами ядер. Рекомендация «СР.1. Допустите, что ваш код будет выполняться в составе многопоточной программы» в Core Guideline подтверждает эту истину.

ПРЕДОТВРАЩЕНИЕ ВЗАИМОБЛОКИРОВОК И ГОНКОВ ЗА ДАННЫМИ

Взаимоблокировки и состояния гонки за данными вызваны нарушениями очередности доступа. Два потока конкурируют за один ресурс: за доступ к части данных. Именно это — причина появления данной рекомендации. В отсутствие совместного доступа к данным для записи не возникает проблем гонки и взаимоблокировки. Но, по сути происходящего, потоки выполнения должны взаимодействовать друг с другом. Поэтому необходим такой способ чтения и записи данных, который гарантировал бы отсутствие вмешательства со стороны других потоков.

В стандартной библиотеке C++11 появился новый тип `std::atomic`, поддерживающий это. В частности, если один поток только пишет данные

в объект `atomic`, а другой поток только читает из него, то в этом случае нет места для неожиданностей и с миром все в порядке. Но что произойдет, если двум потокам одновременно потребуется записать данные в этот объект? Теперь нам нужен флаг, чтобы предупредить другие потоки о том, что в объект производится запись данных, и никакие другие потоки не должны пытаться писать в него. И тут встает похожая проблема: как узнать, можно ли выполнить запись во флаг? Два потока могут одновременно попытаться изменить состояние флага и все испортить.

К счастью, существует понятие мьютекса. Мьютекс — это абстракция флага, доступного только одному потоку. В C++11 появился библиотечный тип `std::mutex`, помогающий обеспечить исключительный доступ к чему-либо. Его API прост: есть две функции блокировки, `lock()` и `try_lock()`, и одна функция разблокировки, `unlock()`. Если потоку удалось заблокировать мьютекс, никакой другой поток не сможет сделать то же самое.

Задача решена! Получается, что теперь не о чем беспокоиться? К сожалению, довольно легко оказаться в ситуации, когда два потока будут ждать друг друга, чтобы освободить мьютекс.

```
void thread1()
{
    // Заблокировать мьютекс записи
    // Заблокировать мьютекс журнала
    // Выполнить запись
    // Записать сообщение в журнал
    // Разблокировать мьютекс записи
    // Разблокировать мьютекс журнала
}

void thread2()
{
    // Заблокировать мьютекс журнала
    // Заблокировать мьютекс записи
    // Выполнить запись
    // Записать сообщение в журнал
    // Разблокировать мьютекс записи
    // Разблокировать мьютекс журнала
}
```

Если оба потока начнут выполняться одновременно, то поток 1 может заблокировать мьютекс записи, а поток 2 — мьютекс журнала. Затем оба потока будут ждать, пока другой поток разблокирует свой мьютекс.

«Это глупо, — скажете вы. — Они должны просто заблокировать нужный им мьютекс, выполнить соответствующие операции и разблокировать его, прежде чем заблокировать следующий мьютекс».

```
void thread1()
{
    // Заблокировать мьютекс записи
    // Выполнить запись
    // Разблокировать мьютекс записи
    // Заблокировать мьютекс журнала
    // Записать сообщение в журнал
    // Разблокировать мьютекс журнала
}

void thread2()
{
    // Заблокировать мьютекс записи
    // Выполнить запись
    // Разблокировать мьютекс записи
    // Заблокировать мьютекс журнала
    // Записать сообщение в журнал
    // Разблокировать мьютекс журнала
}
```

«Теперь все должно работать правильно, а рекомендация гласит: “Не блокируйте более одного мьютекса за раз”. Честно говоря, и я мог бы написать так».

Безусловно, это была бы полезная рекомендация, но как бы вы ее применяли? А если функция записи заблокирует другой мьютекс? Может быть, вы предполагаете, что нам нужен какой-то особый способ оформления функций, наподобие ключевого слова `const`, не блокирующий мьютексы? Но не станет ли это довольно серьезным ограничением для разработки многопоточных программ? На самом деле в Core Guideline есть еще одна рекомендация: «СР.22. Никогда не вызывайте неизвестный код, удерживая блокировку (например, обратный вызов)». Эта рекомендация помогает разрешить описанную ситуацию.

К сожалению, эта проблема не имеет решения. Вы можете вызвать `std::lock()` или использовать класс `std::scoped_lock`, но стиль программирования, основанный на совместном использовании данных для записи, — это путь к катастрофе, поэтому вы должны минимизировать такие ситуации, если не удается избежать их полностью. На самом деле инфраструктура мьютексов решает совсем другую проблему: мы должны не общаться, делясь памятью, а делиться памятью, общаясь.

ОТКАЗ ОТ БЛОКИРОВОК И МЬЮТЕКСОВ

Фраза «Не общайтесь, делясь памятью; делитесь памятью, общаясь» — первый девиз языка Go. Эта мудрость так же стара, как конкуренция. Применительно к C++ она требует полного отказа от мьютексов и связанных с ними данных и использования другого способа взаимодействий между потоками, например, путем передачи сообщений.

Сообщения — это небольшие объекты, передаваемые по значению в очереди сообщений. Каждый поток будет поддерживать очередь, периодически проверять ее на наличие новых сообщений и реагировать на эти сообщения соответствующим образом. Когда один поток выполнил свою часть работы и ему нужно сообщить другому потоку, что доступны результаты его деятельности, он посыпает сообщение. Ссылку на результат можно передать в объекте `std::unique_ptr`, а не в объекте `std::shared_ptr`, потому что применение последнего противоречит сути подхода.

Конечно, мы просто переместили проблему в другое место. Так или иначе, мы не можем полностью отказаться от совместного использования некоторых данных, но здесь мы абстрагировали их в объект очереди. Объект очереди — это единственное место, где будут происходить манипуляции с совместным использованием записей, и это может избавить нас от риска выстрелить себе в ногу.

Используя объекты `std::unique_ptr` в качестве ссылок на данные и перемещая их из потока в поток через очередь сообщений¹, мы можем рассматривать каждый поток как автономную единицу работы и свести к минимуму объем общих данных для записи, необходимых для управления очередью.

А теперь обобщим все сказанное выше:

- общие данные, доступные для записи, могут привести к гонке за данными и взаимоблокировкам;
- правильная синхронизация — сложная задача;
- минимизируйте использование общей памяти, например обмениваясь сообщениями.

¹ На момент написания книги в библиотеке не было стандартной конкурентной очереди, хотя документ, описывающий ее, находится в разработке уже несколько лет: www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0059r4.pdf.

Мы достигли конца обсуждения рекомендации, но это еще не конец главы. Ведь начато путешествие по абстракциям, если помните, которое мы еще не закончили.

Нашей первой абстракцией был объект `atomic`, не допускающий возможности одновременного изменения его значения двумя потоками. Следующей абстракцией был мьютекс: атомарный объект, служащий флагом, который разрешает или запрещает доступ к ресурсу. Третьей абстракцией была очередь сообщений: механизм безопасного обмена данными между потоками. Наша последняя абстракция превращает потоки в задачи и является темой рекомендации «СР.4. Размышляйте с точки зрения задач, а не потоков выполнения».

Рассмотрим предыдущий пример заполнения буфера данными, получаемыми из устройства ввода/вывода. Основную сложность, связанную с многопоточностью, представляет вопрос совместного использования памяти. Если поток полностью изолирован от остальной части программы, то никаких проблем не должно возникать. Поток легко сделать безопасным: просто не открывайте доступ к заполняющемуся буферу, пока он не будет заполнен. Задачу можно описать так: «получить имя файла и вернуть буфер, заполненный содержимым этого файла». Ее легко инкапсулировать в класс, например:

```
class file_contents
{
public:
    file_contents(std::string const& filename);
    ~file_contents();
    std::pair<std::byte*, size_t> buffer() const;

private:
    ...
};
```

Конструктор класса получает имя файла, находит его, определяет размер, создает буфер и запускает поток выполнения, запрашивая уведомление о его завершении. Пока буфер не заполнен, функция `buffer()` возвращает `{nullptr, 0}`. После заполнения она будет возвращать адрес буфера и количество байтов в нем. Обработка ошибок нередко оказывается сложной задачей: вы можете предпочесть сгенерировать исключение при вызове `buffer()`, чтобы сообщить об ошибках, или добавить предикат `good()`.

Существует также приватная реализация, которую следует учитывать, но и без этого совершенно ясно, что в данном случае нет места для гонки за

данными или взаимоблокировок: доступ к буферу возможен только после завершения ввода/вывода, и нет никаких мьютексов, которые могли бы вызвать проблемы.

Более сложный пример — типичная игра для телефона, консоли или ПК. Во время игры важно помнить о разных аппаратных компонентах. Игра получает данные из контроллера и, возможно, из сетевого порта. Извлекает графические данные, аудиоданные и текст из автономного хранилища. Запускает модель мира, в котором протекает игра, и обновляет ее с определенной частотой. Все эти задачи выполняются одновременно. В каждой итерации игрового цикла модель принимает и обрабатывает входные данные, а тем временем видеодвижок рисует ее моментальный снимок на экране, а аудиодвижок воспроизводит звуковые эффекты. Задача, отвечающая за ввод данных, отправляет их задаче моделирования, которая пересыпает данные движкам. Все это — задачи.

Рассмотрим теперь двухпроцессорную рабочую станцию: операционная система может рассматривать каждый процесс как отдельную задачу, не взаимодействующую с другими задачами (кроме разве что буфера обмена, динамического обмена данными (Dynamic Data Exchange, DDE), связывания и внедрения объектов (Object Linking and Embedding, OLE) и т. д., которые реализуются и поддерживаются операционной системой). Если задача потерпела неудачу, система от этого не пострадает. Она не зависит ни от одной из задач.

Создание и уничтожение потоков может оказаться дорогим удовольствием. Лучше всего пересыпать сообщения между потоками, используя пул потоков, которые просто пристаивают в ожидании. Периодически пул передается пакет инструкций и данных, после чего он запускает один из ожидающих потоков в работу. Поток выполняет инструкции, используя данные, возвращает значение отправителю и снова приостанавливается.

Этот подход не только экономит на накладных расходах, связанных с созданием и уничтожением потоков, но и обеспечивает высокую масштабируемость. Когда эксперименты с конкурентными вычислениями еще только начинались, в распоряжении разработчика имелось, скажем, два одноядерных процессора. В настоящее время доступны процессоры с количеством ядер, выражаемым двузначным числом. Эффективное использование всех этих ядер возможно только путем полного разложения проблемы на задачи, которые можно решать независимо друг от друга.

Секрет в том, чтобы программисту перестать думать в терминах потоков выполнения, которые являются частью области решений, и начать думать

в терминах задач, являющихся частью предметной области. В частности, в Core Guidelines говорится: «Поток — это концепция реализации, представляющая физическую машину. Задача — это прикладная концепция, представляющая некоторое задание, которое необходимо выполнить, желательно одновременно с другими задачами. Рассуждать с использованием прикладных концепций проще».

Задачи — это суть вашей программы, набор целей, которые ваша программа должна достичь. Сделайте свои задачи независимыми и опишите их как наборы инструкций и данных. Рекомендация «СР.4. Размышляйте с точки зрения задач, а не потоков выполнения» решает эту проблему. Стандарт в конечном итоге добавит пулы потоков в библиотеку поддержки конкурентного выполнения, после чего вам вообще не придется думать о потоках. Но до тех пор привлекайте потоки только на соответствующем уровне абстракции и не используйте записываемых общих данных.

ПОДВЕДЕМ ИТОГ

При разработке многопоточных приложений важно помнить о нескольких аспектах. Конкурентное программирование по-прежнему остается самой сложной областью разработки программного обеспечения на C++. Гонки за данными и взаимоблокировки — это сложные проблемы, и переход к использованию концепции задач значительно облегчит вашу жизнь. Но до тех пор:

- остерегайтесь захватывать сразу несколько мьютексов;
- создавайте слабо связанные абстракции, отражающие задачи, которые нужно выполнить;
- сводите к минимуму необходимость использования низкоуровневых абстракций из библиотеки поддержки конкурентного выполнения.

ГЛАВА 2.6

Т.120. Используйте метапрограммирование шаблонов, только когда это действительно необходимо

Выбор докладов для представления на конференции по C++ осуществляется в программном комитете в форме голосования: обычно комитет насчитывает около дюжины членов. Одной из тем докладов, гарантированно получавших высокий балл, еще несколько конференций назад была тема метапрограммирования шаблонов, которую далее мы будем называть TMP (template metaprogramming). Это захватывающая и многообещающая область разработки на C++, но частенько она приносит больше вреда, чем пользы: по сути, она представляет собой еще одно проявление синдрома блестящего молотка.

Есть несколько особенностей, делающих TMP сложной и привлекательной темой для организаторов конференций и разработчиков, но отталкивающей для инженеров-менеджеров.

Во-первых, метапрограмма выполняется во время компиляции, но во время компиляции нет доступного изменяемого состояния. Из-за этого метапрограммирование очень похоже на функциональное программирование — сложную парадигму для освоения.

Во-вторых, отсутствует возможность управления потоком, кроме как посредством рекурсии. Однако, чтобы понять суть рекурсии, требуется

приложить некоторые усилия. Существует даже выражение «Итерации — от человека, рекурсия — от Бога».

В-третьих, отладка метапрограмм невозможна с использованием обычных средств. Если код не работает, то вы должны просто просматривать его снова и снова, пока не наступит озарение. Других вариантов нет. Если вам повезет, вы получите сообщение об ошибке компилятора, которое может породить массу других сообщений длиной в сотни символов.

В-четвертых, нужно быть чрезвычайно милосердными, чтобы описать часть TMP как самодокументирующуюся. Конструкции, используемые в метапрограммировании, непрозрачны и требуют досконального понимания некоторых пыльных уголков языка и стандартной библиотеки.

В-пятых, время компиляции может значительно увеличиться. Для создания экземпляра шаблона требуется время, и если это происходит внутри заголовочного файла, то затраты времени могут увеличиться многократно. Конечно, для работы шаблонов функций и классов требуются встроенные определения, так что это вполне обычная ситуация.

К сожалению, трудно не поддаться программистскому азарту от того, что в каких-то случаях применение TMP дает положительный результат. Кроме того, TMP может предложить превосходное представление предметной области в форме обобщенных решений, пригодных для многократного использования. Весьма заманчиво обратиться к TMP, когда в этом нет необходимости.

Но что такое метапрограммирование и как его моделирует TMP?

В двух словах, метапрограммирование стирает грань между состоянием и выполнением, позволяя рассматривать код как данные. Программа может видеть и понимать свой собственный код, анализировать его и принимать решения на основе своей собственной структуры и содержимого. Кроме всего прочего, это приводит к созданию самомодифицирующегося кода. Например, один из авторов в предыдущих главах описывает, как много лет тому назад знание ассемблера Z80 позволило ему изменять код прямо во время работы этого кода. Он мог менять адреса переходов во время выполнения, чтобы менять поведение алгоритмов в процессе работы.

Компилятор C++ знает язык и обрабатывает все объявления как данные. Он использует эти данные для создания окончательной программы. То есть

имеются возможности для метапрограммирования, реализованные через шаблоны. Шаблоны — это точки настройки языка, дающие новые типы в зависимости от особенностей их создания. Эти типы можно рассматривать как результат вычислений на основе ввода, предоставленного программистом. Например, при наличии объявления шаблона функции:

```
template <class T> void fn(T);
```

объявление:

```
fn<int>(17);
```

можно рассматривать как результат применения `int` к шаблону функции. Так C++ превращается в свой собственный метаязык, в средство интроспекции. Как будет показано далее, средства интроспекции, предлагаемые языком C++, несмотря на их полноту по Тьюрингу, скорее запутывают код, чем проясняют его.

Вот тривиальный пример вычисления суммы целых чисел от 1 до `N`:

```
// Рекурсивный алгоритм, обеспечивающий итерации
template <int N> struct sum_integers {
    static constexpr int result = N + sum_integers<N-1>::result;
};

// Явная специализация для формирования терминальной ветви рекурсии
// Обратите внимание, что также достаточно было бы
// использовать if constexpr
template <> struct sum_integers<1> {
    static constexpr int result = 1;
};

int main () {
    return sum_integers<10>::result;
}
```

В примере функция `main` просто вернет 55 — значение, вычисленное во время компиляции. Можете поэкспериментировать и определить, насколько большим может быть значение `N` до того, как компиляция завершится ошибкой. Заодно посмотрите причину ошибки.

Состояние хранится в члене `result` и является константным значением. Повторное явное создание экземпляров `sum_integers` с последовательно уменьшающимися значениями параметра обеспечивает требуемую рекурсию. Если закомментировать терминальную ветвь, то компилятор должен сообщить об ошибке, связанной со сложностью контекста создания

экземпляра. Вот пример вывода, немного отредактированный для наглядности, полученный от компилятора clang 12.0.0:

```
<source>:3:37: fatal error:  
recursive template instantiation exceeded maximum depth of 1024  
    static constexpr int result = N + sum_integers<N-1>::result;  
                                         ^  
  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<-1014>' requested here  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<-1013>' requested here  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<-1012>' requested here  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<-1011>' requested here  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<-1010>' requested here  
<source>:3:37: note:  
  (skipping 1015 contexts in backtrace;  
   use -ftemplate-backtrace-limit=0 to see all)  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<6>' requested here  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<7>' requested here  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<8>' requested here  
<source>:3:37: note:  
  in instantiation of template class 'sum_integers<9>' requested here  
<source>:12:10: note:  
  in instantiation of template class 'sum_integers<10>' requested here  
return sum_integers<10>::result;  
                                         ^  
  
<source>:3:37: note:  
  use -ftemplate-depth=N to increase recursive template instantiation depth  
static constexpr int result = N + sum_integers<N-1>::result;  
                                         ^  
  
1 error generated.  
Compiler returned: 1
```

А вот пример вывода компилятора GCC 11.1:

```
<source>:  
In instantiation of 'constexpr const int sum_integers<-889>::result':  
<source>:3:56:  
  recursively required from 'constexpr const int sum_integers<9>::result'  
<source>:3:56:  
  required from 'constexpr const int sum_integers<10>::result'  
<source>:12:28:  
  required from here  
<source>:3:56: fatal error:
```

```
template instantiation depth exceeds maximum of 900
(use '-ftemplate-depth=' to increase the maximum)
  3 |     static constexpr int result = N + sum_integers<N-1>::result;
  |                                         ^
compilation terminated.
Compiler returned: 1
```

Время работы компилятора MSVC в Compiler Explorer превысило максимальное время ожидания, хотя, к его чести, он дал лучший результат при компиляции правильного исходного кода:

```
main PROC
    mov      eax, 55
    ret      0
main ENDP
```

Если вы знаете, что такое рекурсия и терминальные ветви, или знакомы с доказательством по индукции, то этот результат должен быть вам понятен; если нет, то обращайтесь к документации.

Существует множество ресурсов, демонстрирующих использование частичной специализации для условной компиляции и типов для возврата значений. Эти две возможности вместе с поддержкой итераций через рекурсию дают метапрограммированию шаблонов полноту по Тьюрингу, что является откровенно ужасающей идеей.

Приведенный пример тривиален и прост, поэтому рассмотрим более реалистичный вариант использования: шаблоны выражений¹. Они иногда находят применение в реализациях алгоритмов линейной алгебры. Возьмем следующий класс векторов (здесь под вектором подразумевается математический объект, а не стандартный контейнер). Он может использоваться, например, в трехмерной геометрии. Назовем его `vector_f`, потому что это вектор чисел с плавающей точкой:

```
template <size_t N>
class vector_f {
public:
    vector_f();
    vector_f(std::initializer_list<float> init);
    float operator[](size_t i) const; // доступ к элементам только для чтения
    float& operator[](size_t i); // доступ к элементам для чтения/записи
```

¹ Авторы рады отдать должное коллеге по комитету Давиду Вандевурду (Daveed Vandevoorde), а также Todd Veldhuizen, отмеченным в статье в «Википедии» https://en.wikipedia.org/wiki/Expression_templates.

```

size_t const size(); // возвращает параметр размера
private:
    std::array<float, N> data;
};

```

Нам нужна возможность складывать объекты `vector_f`, а значит, понадобится шаблон функции, реализующей оператор сложения:

```

template <size_t N>
vector_f<N> operator+(vector_f<N> const& u, vector_f<N> const& v) {
    vector_f<N> sum;
    for (size_t i = 0; i < N; i++) {
        sum[i] = u[i] + v[i];
    }
    return sum;
}

```

Здесь нет ничего сложного. Это простой цикл, но хотелось бы кое-что подчеркнуть: код создает возвращаемое значение, заполняет его и возвращает. Компилятор может выполнить развертывание цикла для небольших значений `N`. Мы не можем инициализировать возвращаемое значение с помощью списка инициализаторов конструктора, но и этого вполне достаточно.

Типичный случай при работе с векторами — сложить сразу несколько векторов в одной операции:

```
vector_f<3> v = a + b + c;
```

В результате получится код с двумя циклами, создающий и отбрасывающий временный объект с результатом `a + b`. С увеличением `N` затраты на эти действия будут становиться все более очевидными. Решить проблему можно, отложив генерацию оператора сложения, насколько это возможно. Оператор сложения может возвращать специальный тип, вычисляющий сумму по запросу, а не немедленно. Это своего рода отложенные (ленивые) вычисления. А теперь следите внимательно...

Прежде всего, нужен класс выражения:

```

template <struct E> class vector_expression {
public:
    float operator[](size_t i) const {
        return static_cast<E const&>(*this)[i];
    }
    size_t size() const {
        return static_cast<E const&>(*this).size;
    }
};

```

Здесь используется необычный рекуррентный шаблон — первая часть TMP. Он делегирует выполнение действий оператору квадратных скобок и методу `size` класса, который является параметром этого шаблона.

Теперь выведем `vector_f` из этого класса:

```
template <size_t N> class vector_f
    : public vector_expression<vector_f<N>> {
public:
    vector_f();
    vector_f(std::initializer_list<float> init);
    template <class E>
    vector_f (vector_expression <E> const& e);
    float operator[](size_t i) const; // доступ к элементам только для чтения
    float& operator[](size_t i); // доступ к элементам для чтения/записи
    size_t const size(); // возвращает параметр размера

private:
    std::array<N, float> data;
};
```

Мы добавили новый конструктор, который принимает родителя в качестве параметра. Здесь создается экземпляр на основе типа выражения, а не списка значений. Вот как это выглядит:

```
template <size_t N>
template <class E>
vector_f<N>::vector_f(vector_expression<E> const& e)
    : data(e.size()) {
    for (size_t i = 0; i != e.size(); ++i) {
        data[i] = e[i]; // ❶
    }
}
```

Здесь выполняются фактические вычисления. Последняя часть — это фактический класс выражений сложения:

```
template <class E1, class E2> class vector_sum
    : public vector_expression<vector_sum<E1, E2>> {
    E1 const& u;
    E2 const& v;

public:
    vector_sum(E1 const& u, E2 const& v);
    float operator[](size_t i) const { return u[i] + v[i]; } // ❷
    size_t size() const { return v.size(); }
};
```

```
template <typename E1, typename E2>
vector_sum<E1, E2> operator+(vector_expression<E1> const& u,
                           vector_expression<E2> const& v) {
    return vector_sum<E1, E2>(*static_cast<E1 const*>(&u),
                               *static_cast<E2 const*>(&v));
}
```

Это все, что необходимо для сложения. Выражение:

`a + b + c`

теперь вместо типа `vector_f<3>` имеет тип:

```
vector_sum<vector_sum<vector_f<3>, vector_f<3>>>
```

Когда это выражение назначается объекту `vector_f<3>`, вызывается конструктор, принимающий `vector_expression`, который присваивает элементы выражения элементам данных ❶. Оператор квадратных скобок для `vector_sum` возвращает сумму двух объектов ❷, рекурсивно превращающуюся в сумму двух других объектов `vector_sum`, которая наконец развертывается в сумму трех элементов.

Как видите, для вычислений не нужны временные объекты и достаточно одного цикла. Этот вид отложенных вычислений также используется в библиотеке ranges: типы выражений конструируются во время компиляции и вычисляются в момент присваивания.

Рассмотренный пример потребовал довольно много пояснений. Очевидно, что и в документации потребуется уделить особое внимание объяснению происходящего. Но самая большая проблема, с которой часто сталкиваются специалисты при проверке кода, — это как раз отсутствие документации. Соответственно, весьма маловероятно, что подобные трюки будут достаточно полно задокументированы. Тем не менее TMP — популярный прием, он может принести пользу, поэтому если вы собираетесь его использовать (конечно, только в том случае, когда это действительно нужно), то сопроводите его подробным описанием. Кроме того, проверьте еще раз, действительно ли он нужен. Авторы компиляторов — умные люди. Убедитесь, что без применения приемов TMP производительность действительно ниже. Посмотрите на генерированную сборку до и после применения. Измерьте производительность. Измерьте время сборки. Прием, дававший преимущество три года назад, может перестать давать его, поэтому будьте осмотрительны, проверяйте, прежде чем принять окончательное решение.

STD::ENABLE_IF => REQUIRES

Когда появился C++98, я¹ первым делом посмотрел, как контейнеры взаимодействуют с алгоритмами. В то время я работал в игровой компании в Лондоне и пытался убедить некоторых инженеров, что C++ — это то, что нужно (возможно, я несколько опережал события). Я продемонстрировал им очень короткие функции, способные выполнять поиск и сортировку коллекций. Однако в ту пору тип `std::vector` был проблемным. Всякий раз, когда происходило превышение его текущей емкости и запускалась процедура изменения размера, его содержимое копировалось поэлементно в новое место с помощью цикла `for`, а затем старые версии удалялись. Конечно, такое решение было обусловлено требованием стандарта: новые значения должны создаваться на новом месте вызовом конструктора копирования, а прежние — уничтожаться вызовом их деструкторов. Однако в большинстве ситуаций вполне подошла бы очевидная оптимизация, заключавшаяся в использовании `memcp` вместо поэлементного копирования.

`memcp` — это функция C, которую никогда не следует использовать напрямую, так как она просто копирует содержимое памяти в указанное место без вызова конструктора копирования. Это означает, что, несмотря на сохранность данных, любая дополнительная инициализация, скажем регистрация объектов в службе уведомлений, и действия, связанные с уничтожением, например отмена регистрации в службе уведомлений, не будут выполнены. `memcp` — это деталь реализации, не предназначенная для повседневного использования.

Мои коллеги послушали меня, развернулись и ушли. Я был удручен и решил написать класс векторов, использующий `memcp` вместо конструктора копирования и деструктора, как это должно было быть. Я назвал его `mem_vector`. Этот класс предназначался для использования только с типами, не имеющими конструкторов или деструкторов, и он получился на славу. Все были в восторге от него и нашли ему хорошее применение в сочетании с алгоритмами. Я был на седьмом небе примерно три дня, пока кто-то все не испортил, добавив поддержку конструктора. Однако `mem_vector` не был предназначен для этого, и вскоре стало ясно, что он не может использоваться в этой ситуации, пока код не будет доработан, а это означало возврат на старый путь.

В чем я *действительно* нуждался, так это в способе выбора между использованием `memcp` и копированием с последующим уничтожением каждого

¹ Экскурс в историю с Гаем Дэвидсоном. Рассмотрим еще один популярный прием.

элемента в отдельности. Для воплощения идеи пришлось приложить некоторые усилия, потому что понадобилось реализовать перегруженные функции-члены, которые выбирались бы для семейств типов, а не каких-то конкретных. Я хотел иметь возможность объявить (напомню, что это было еще до появления C++11):

```
template <class T>
void mem_vector<T>::resize(size_type count);
```

вместе с:

```
template <class Trivial>
void mem_vector<Trivial>::resize(size_type count);
```

и пусть бы компилятор сам выбирал реализацию `mem_vector`, соответствующую характеру параметра шаблона, а не конкретному его типу. Однажды я услышал о SFINAE (Substitution Failure Is Not An Error — ошибка подстановки не является ошибкой), и внезапно меня осенило: я могу встроить пустую структуру с именем `trivial` во все соответствующие типы. Если бы это удалось, я мог бы объявить вторую функцию `resize` так:

```
template <class T>
void mem_vector<T>::resize(size_type count, T::trivial* = 0);
```

Если бы структура-член `trivial` не существовала, то функция не рассматривалась бы. Но это решение просто перемещает проблему в другое место. По-прежнему не было никакого способа принудительно удалить структуру-член `trivial`, если класс переставал быть тривиальным. Эта проблема была особенно характерна для случаев структур, наследующих от других тривиальных структур, когда родительская структура становится нетривиальной. Я впал в отчаяние.

Но все оказалось не так плохо: мы вместе смогли кое-что придумать, используя встроенные функции, характерные для компилятора, а затем появился C++11. Внезапно проблема была решена благодаря удивительной мощи `std::enable_if` и заголовку `type_traits`. О чудный день!

Однако была одна маленькая проблема. Код был почти нечитаемым. Такие типы, как:

```
std::enable_if<std::is_trivially_constructible<T>::value>::type
```

(да-да, это тип) были разбросаны по всему коду, создавая дополнительную когнитивную нагрузку.

Проблема с этим приемом TMP заключается в том, что он запутывает код и не позволяет быстро понять происходящее. Честно говоря, для меня это больше похоже на шум. Да, накопив опыт, мы научились читать и понимать такой код, но новым программистам приходилось проходить долгий период обучения.

Однако сложность этого приема не помешала широкому его внедрению. Поискав по старым репозиториям на GitHub, можно быстро найти фрагменты кода с тщательно продуманными перегрузками функций, отличающимися предложениями `std::enable_if` с различной степенью детализации. Комитет не остался глух к этой проблеме, и в C++17 была добавлена новая возможность — оператор `if constexpr`.

Он стал избавлением от нескольких классов проблем, которые исторически заставляли использовать `enable_if`. С его помощью можно оценить выражение во время компиляции и выбрать подходящее исполнение. Например:

```
if constexpr(sizeof(int) == 4)
{
    // тип int имеет размер 32 бита
}
```

В частности, его можно использовать там, где прежде нужно было использовать `enable_if`:

```
if constexpr(std::is_move_constructible_v<T>)
{
    // шаблон функции, специализированный для перемещаемого типа
}
```

Вместо перегрузки шаблонов функций для разных типов теперь можно выразить разницу в одной функции. Например, следующую пару переопределенных функций:

```
template <class T, typename =
    std::enable_if<std::is_move_constructible_v<T> >::type>
void do_stuff()
{
    ...
}

template <class T, typename =
    std::enable_if<!std::is_move_constructible_v<T> >::type>
void do_stuff()
{
    ...
}
```

можно заменить на:

```
template <class T>
void do_stuff()
{
    if constexpr(std::is_move_constructible_v<T>)
    {
        ...
    }
    else
    {
        ...
    }
}
```

Однако комитет на этом не остановился. После продолжительного периода обсуждения в язык были добавлены понятия, которые принесли с собой ограничения и предложения `requires`. Это значительно упростило определение ограничений на типы для специализации. Вот как выглядит предложение `requires`:

```
template <class T>
requires std::is_move_constructible_v<T>
void do_stuff()
{
    ...
}
```

Эта функция доступна только для типов, имеющих конструктор перемещения. Концептуально этот подход идентичен `std::enable_if`, но намного проще и понятнее. Он дает возможность избежать TMP и четко обозначить намерения.

Но и это еще не все. Одна из исследовательских групп комитета — SG7 — занимается вопросами интроспекции, которая в значительной степени осуществляется с применением TMP. Цель этой группы — добавить в язык средства, созданные с использованием TMP, и устраниТЬ наконец необходимость в TMP как таковом. Интроспекция — важная часть метапрограммирования, и исследовательская группа пытается соединить множество идей, чтобы выработать согласованную стратегию и решение проблем метапрограммирования. Разрабатываются дополнительные возможности, основанные на интроспекции, одна из которых — метаклассы. Они должны позволить программистам определять не только подставляемые типы, но и форму классов, чтобы дать клиентам метакласса возможность создавать экземпляры, не заботясь о стандартном коде.

Все эти возможности, как предполагается, повысят ясность кода: вместо добавления дополнительных понятий, требующих изучения и освоения, они позволяют упростить большую часть существующего кода и устранит ненужные угловые скобки.

Мы надеемся, что Руководство убедило вас в сложности ТМР, объяснило, почему его следует использовать только в крайнем случае. Конечно, иногда метапрограммирование возможно, но только при вдумчивом и осторожном использовании шаблонов.

Существует афоризм, который каждый инженер должен держать близко к сердцу: «Умный код — это просто. Простой код — это умно».

Лучшим считается код, взглянув на который читатель скажет: «Здесь нет ничего особенного. Все очевидно». Однако в адрес метапрограммирования шаблонов такое можно услышать чрезвычайно редко.

ПОДВЕДЕМ ИТОГ

- Метапрограммирование в C++ моделируется с помощью шаблонов функций и классов.
- Анализ, проводимый комитетом по C++, предлагает интроспекцию.
- Наиболее полезные приемы метапрограммирования были явно перенесены в язык.
- Эта тенденция продолжается и направлена на устранение необходимости в метапрограммировании шаблонов.



ПРЕКРАТИТЕ ЭТО ИСПОЛЬЗОВАТЬ

Глава 3.1 I.11. Никогда не передавайте владение через простой указатель (T^*) или ссылку ($T\&$).

Глава 3.2 I.3. Избегайте синглтонов.

Глава 3.3 C.90. Полагайтесь на конструкторы и операторы присваивания вместо `memset` и `memcp`.

Глава 3.4 ES.50. Не приводите переменные с квалификатором `const` к неконстантному типу.

Глава 3.5 E.28. При обработке ошибок избегайте глобальных состояний (например, `errno`).

Глава 3.6 SF.7. Не используйте `using namespace` в глобальной области видимости в заголовочном файле.

ГЛАВА 3.1

I.11. Никогда не передавайте владение через простой указатель (T^*) или ссылку ($T\&$)

ИСПОЛЬЗОВАНИЕ ОБЛАСТИ СВОБОДНОЙ ПАМЯТИ

Владение — важное обстоятельство. За ним стоит ответственность, под которой в C++ подразумевается уборка и освобождение ресурсов за собой. Если вы что-то создали, то по завершении всех процессов должны это уничтожить и освободить память. Для объектов со статическим и автоматическим классами хранения это совершенно тривиальный вопрос, но для объектов, динамически размещаемых в области свободной памяти, это настоящее минное поле.

Память, выделенную из свободной области, легко потерять. Ее можно вернуть только с помощью указателя, назначенного этой выделенной памяти, который является единственным доступным дескриптором. Если указатель покинет область видимости, не будучи скопированным, то освободить память не удастся. Такое явление известно как утечка памяти. Например:

```
size_t make_a_wish(int id, std::string owner) {
    Wish* wish = new Wish(wishes[id], owner);
    return wish->size();
}
```

По завершении функции указатель `Wish` покидает область видимости, что делает невозможным освобождение памяти. Мы можем немного изменить

функцию и вернуть из нее указатель, чтобы вызывающая сторона могла взять на себя ответственность и удалить объект позже, освободив память.

```
Wish* make_a_wish_better(int id, std::string owner) {  
    Wish* wish = new Wish(wishes[id], owner);  
    return wish;  
}
```

Это правильно оформленный код, хотя мы бы не назвали его современным в идиоматическом смысле. К сожалению, он провоцирует возникновение определенных сложностей: вызывающий обязан стать владельцем объекта и гарантировать его уничтожение с помощью оператора `delete` по окончании использования. Также существует опасность удалить объект до того, как он действительно перестанет быть нужным. Если `make_a_wish` получит указатель от другого объекта, то как сообщить этому другому объекту, что указатель больше не нужен?

Исторически функции такого типа приводили к истощению области свободной памяти из-за размещения объектов-зомби, право собственности на которые не было четко обозначено и которые никогда не удалялись. Передачу права собственности можно оформить несколькими способами. Автор функции, назвав ее `allocate_a_wish`, может тем самым подсказать клиенту, что для результата была выделена память и теперь вся ответственность за освобождение этой памяти возлагается на того, кто функцию вызывал.

Это довольно слабый способ обозначения передачи права владения. Он не является обязательным к исполнению, и многое зависит от того, помнит ли клиент об ответственности и действует ли надлежащим образом. Также требуется, чтобы автор внедрил реализацию в интерфейс. А это уже просто плохая привычка, потому что неявно раскрывает детали реализации клиенту и не позволяет вам изменить их без риска привнесения заблуждений.

Подобное именование может показаться слабым решением, но в действительности оно не настолько слабо, как, скажем, упоминание о необходимости освобождать память, размещенное в документации на сервере в каком-то удаленном, темном, потаенном месте. И не настолько слабо, как комментарии в заголовочном файле, которые никто никогда не читает. Но даже притом, что это меньшее зло, такой подход все же ненадежен.

Еще хуже возврат значения через ссылку вместо указателя. Как вызывающий код узнает, что объект был уничтожен? Имея такое значение, он может лишь надеяться, что другой поток не уничтожит его между делом,

и может, конечно, убедиться, что объект используется, прежде чем вызывать другую функцию, которая способна инициировать его уничтожение. Такой подход требует учитывать слишком много контекста при разработке.

Работающие с особенно старым кодом могут увидеть экземпляры `std::auto_ptr`. Это первая попытка решить данную проблему, которая в итоге была стандартизована в C++98. Тип `std::auto_ptr` содержит сам указатель и обеспечивает семантику перегруженного указателя, действуя как дескриптор объекта. Экземпляры `std::auto_ptr` можно передавать, освобождая от права собственности при копировании. Выходя из области видимости при сохранении за ними прав собственности, они удаляют хранимые в памяти объекты. Однако необычная семантика копирования означает, что объекты `std::auto_ptr` нельзя без опаски хранить в стандартных контейнерах. Этот класс считался устаревшим уже во втором стандарте (C++11) и был удален в третьем (C++14).

Однако комитет ничего не удаляет из языка без замены, а внедрение семантики перемещения позволило создать объекты, владеющие указателями, для которых хранение в контейнерах не было проблемой. Поскольку `std::auto_ptr` был объявлен устаревшим в C++11, ему на смену были введены типы `std::unique_ptr` и `std::shared_ptr`. Они известны как «умные» или «интеллектуальные» указатели и полностью решают проблему владения.

Получив объект `std::unique_ptr`, код становится владельцем того, на что указывает этот объект. При выходе из области видимости он удаляет хранимый объект. Однако, в отличие от `std::auto_ptr`, `std::unique_ptr` не имеет флага, определяющего право собственности, поэтому его можно безопасно хранить в стандартных контейнерах. Причина, почему ему не нужен флаг, заключается в том, что его нельзя скопировать, его можно только переместить, поэтому нет никакого недопонимания в отношении того, кто в данный момент владеет содержащимся в нем объектом.

Получив объект `std::shared_ptr`, вы начинаете интересоваться тем, на что он указывает. Когда он выходит из области видимости, этот интерес пропадает. Когда объектом больше никто не интересуется, он удаляется. Право собственности распределяется между всеми имеющими отношение к хранимому объекту. Объект не будет уничтожен, пока остается хотя бы один объект, интересующийся им.

По умолчанию для хранения динамических объектов следует использовать `std::unique_ptr`, а тип `std::shared_ptr` должен использоваться, только если рассуждения о времени жизни и владении невероятно запутаны. Но даже

в этом случае применение типа `std::shared_ptr` следует рассматривать как признак технической недоработки, обусловленной несоблюдением соответствующей абстракции. Одним из примеров может служить программа просмотра Twitter, которая распределяет твиты по разным столбцам. Твиты могут содержать изображения, которые делают их большими и заставляют размещаться в нескольких разных столбцах. Твит должен продолжать существовать в памяти программы, только когда он виден в одном из столбцов, но пользователь может в какой-то момент решить, что твит больше не нужен, прокрутив все столбцы так, что он исчезнет отовсюду. Можно сохранить контейнер твитов и использовать счетчики, подсчитывая ссылки на твиты вручную, но это решение просто дублирует абстракцию `std::shared_ptr` на другом уровне косвенности. В частности, вспомним, что решение о сроке жизни твита принимает пользователь, а не программа. Такая ситуация должна быть редкостью.

ПРОИЗВОДИТЕЛЬНОСТЬ ИНТЕЛЛЕКТУАЛЬНЫХ УКАЗАТЕЛЕЙ

Иногда использование интеллектуальных указателей может быть нежелательно. Копирование `std::shared_ptr` не обходится без затрат. Тип `std::shared_ptr` должен быть потокобезопасным, что приводит к дополнительным затратам, отрицательно сказывается на производительности. Потокобезопасным является только блок управления `std::shared_ptr`, но не сам ресурс. Он может быть реализован как пара указателей, один из которых указывает на хранимый объект, а другой — на механизм подсчета. Копирование обходится недорого с точки зрения передачи памяти, но механизм подсчета должен будет получить мьютекс и увеличить счетчик ссылок при его копировании. Когда `std::shared_ptr` выйдет из области видимости, механизму поддержки снова придется получить мьютекс и уменьшить счетчик ссылок, а при обнулении счетчика — уничтожить объект.

По умолчанию для хранения динамических объектов следует использовать `std::unique_ptr`, а тип `std::shared_ptr` должен использоваться, только если рассуждения о времени жизни и владении невероятно запутаны. Но даже в этом случае применение типа `std::shared_ptr` следует рассматривать как признак технической недоработки, обусловленной несоблюдением соответствующей абстракции.

Тип `std::unique_ptr` проще и дешевле. Его можно только перемещать, но не копировать, поэтому может существовать только один его экземпляр. Соответственно, когда экземпляр `std::unique_ptr` покидает область видимости, он должен удалить хранимый объект. Никакого подсчета не требуется. И все же на хранение указателя на функцию, которая удалит объект, расходуется дополнительная память. Тип `std::shared_ptr` тоже содержит такой объект как часть подсчета.

Об этих накладных расходах можно не беспокоиться, пока они не превратятся в горячую точку в отчетах профилировщика. Безопасность интеллектуального указателя — очень ценное обретение. Однако иногда, обнаружив, что применение интеллектуальных указателей отражается на производительности, и посмотрев, куда они передаются, можно заметить, что совместное использование или передача прав собственности вообще не нужны. Например:

```
size_t measure_widget(std::shared_ptr<Widget> w) {
    return w->size(); // (предполагается, что w != null)
}
```

Этой функции не требуется владеть указателем. Она просто вызывает другую функцию и возвращает полученное значение. Следующая функция будет работать так же хорошо:

```
size_t measure_widget(Widget* w) {
    return w->size(); // (предполагается, что w != null)
}
```

Обратите внимание, что произошло с `w` или, скорее, чего не произошло. Указатель не передавался другой функции, не использовался для инициализации другого объекта, и его время жизни никак не было продлено. Если бы функция выглядела так:

```
size_t measure_widget(Widget* w) {
    return size(w); // (Вы правильно подумали...)
}
```

тогда другое дело. Вы не владеете `w`, поэтому не можете им распоряжаться. Функция `size` может создать копию `w` и кэшировать ее для последующего использования. Соответственно, если вы не уверены в реализации этой функции и не отвечаете за возможное ее изменение в будущем, такая передача `w` может оказаться небезопасной. Если объект, на который указывает `w`,

будет уничтожен позже, то копия `w` будет указывать на несуществующий объект и ее разыменование может привести к катастрофе.

Эта функция принимает объект по указателю и передает его другой функции. Такая последовательность действий подразумевает владение, которое не передается в сигнатуре функции. Не передавайте владение через простые указатели.

Правильный способ реализации функции:

```
size_t measure_widget(std::shared_ptr<Widget> w) {
    return size(w);
}
```

Теперь вы даете функции `size()` возможность заявить о своей заинтересованности в `std::shared_ptr`. Если вызывающая функция впоследствии уничтожит `w`, то копия, созданная функцией `size()`, останется действительной.

ИСПОЛЬЗОВАНИЕ ПРОСТОЙ СЕМАНТИКИ ССЫЛОК

Простой указатель не единственный способ передачи объектов не по значению. Для этого также можно использовать ссылки. Использование ссылок — предпочтительный механизм передачи по ссылке. Взгляните на следующую версию `measure_widget`:

```
size_t measure_widget(Widget& w) {
    return w.size(); // (Ссылки не могут быть пустыми,
                    // разве что по злому умыслу)
}
```

Это решение лучше предыдущих, потому что перекладывает бремя проверки существования объекта на вызывающий код. Он должен разыменовать объект и заплатить штраф за разыменование пустого указателя. Однако, если `w` передается дальше, возникает та же проблема с владением, что и раньше. Если ссылка хранится как часть другого объекта, а объект ссылки уничтожается, то ссылка перестает быть действительной.

Сигнатура функции должна сообщать вызывающей стороне все, что та должна знать о владении. Если сигнатура включает `T*`, то вызывающая

сторона может передать указатель на объект или пустой указатель и не беспокоиться о его времени жизни. Вызывающий код просто передает объект по ссылке в функцию, а затем продолжает работу. Если сигнатура включает `T&`, то вызывающая сторона может передать ссылку на объект и не беспокоиться о его времени жизни. Здесь применимы все те же рассуждения.

Если сигнатура включает `std::unique_ptr<T>`, это означает, что вызывающая сторона должна отказаться от владения объектом. Если сигнатура включает `std::shared_ptr<T>`, это говорит о том, что вызывающая сторона должна допускать владение объектом совместно с вызываемой функцией и не может быть уверена в том, когда объект будет уничтожен.

Отклонившись от этих правил, вы рискуете внести в свой код трудноразличимые и болезненные ошибки, которые повлекут утомительные споры о праве собственности и ответственности. В конечном итоге объекты будут или уничтожаться слишком рано, или не уничтожаться вообще. Не передавайте право собственности с использованием простых указателей или ссылок. Если ваша функция принимает указатель или ссылку, не передавайте ее конструктору или другой функции, не осознавая при этом, какая ответственность за это ложится на вас.

GSL::OWNER

Мы рассмотрели передачу и возврат значений по простому указателю и ссылке и отметили, что это не самая лучшая идея. Пользователи могут сделать неверный вывод о праве владения объектом. Они могут захотеть обрести право собственности, когда это не предусмотрено. Решение данной проблемы — использовать интеллектуальные указатели.

К сожалению, работая с устаревшим кодом, вы не всегда можете сильно его изменить. Он может быть частью зависимости другого устаревшего кода, который полагается на ABI. Замена простых указателей интеллектуальными изменит представление объектов в памяти, нарушив ABI.

Настал момент представить вашему вниманию библиотеку поддержки рекомендаций (Guidelines Support Library, GSL). Это небольшая библиотека средств поддержки от Core Guidelines. В самом Руководстве много рекомендаций, которые очень трудно соблюсти. Ярким примером таких трудностей является использование простых указателей: как сообщить о праве владения

в отсутствие возможности использовать интеллектуальные указатели? GSL предоставляет типы, помогающие выполнить эту рекомендацию.

GSL состоит из пяти частей:

- GSL.view — типы в этой части позволяют различать владеющие и не-владеющие указатели, а также указатели на единственный объект и указатели на первый элемент последовательности;
- GSL.owner — указатели с правом владения, включая `std::unique_ptr` и `std::shared_ptr`, а также `stack_array` (массив, размещенный в стеке) и `dyn_array` (массив, размещенный в куче);
- GSL.assert — предвосхищая предложение о контрактах, предоставляет два макроса: `Requires` и `Ensures`;
- GSL.util — ни одна библиотека не обходится без набора разных полезных вещей, и они здесь;
- GSL.concept — коллекция предикатов типов.

GSL появилась до выхода стандарта C++17, и некоторые части GSL, в частности раздел `concept`, были заменены стандартом C++. Библиотека доступна на GitHub по адресу <https://github.com/Microsoft/GSL>. Просто добавьте директиву `#include <gsl/gsl>`, чтобы получить в свое распоряжение полный набор объектов.

Этот раздел в большей степени посвящен одному из типов представлений, `gsl::owner<T*>`. Рассмотрим пример:

```
#include <gsl/gsl>

gsl::owner<int*> produce()          // Превращает вас в счастливого владельца
{
    gsl::owner<int*> i = new int;   // Вы – владелец
    return i;                      // Передача владения из функции
}

void consume(gsl::owner<int*> i) // Прием права владения
{
    delete i;                     // Теперь это ваше, вы можете уничтожить его
}

void p_and_c()
{
    auto i = produce();           // создать...
    consume(i);                  // ...и уничтожить
}
```

Как видите, заключение указателя в `owner< >` сигнализирует о праве собственности. Давайте немного изменим ситуацию:

```
int* produce() // Простой указатель
{
    gsl::owner<int*> i = new int;
    return i;
}
```

Что произойдет в этом случае?

Не следует ждать, что компилятор предупредит вас о преобразовании объекта с информацией о владении в простой указатель. К сожалению, это не тот случай. Взгляните, как определен тип `owner`:

```
template <class T,
          class = std::enable_if_t<std::is_pointer<T>::value>>
using owner = T;
```

Как видите, здесь нет никакой магии. Тип `gsl::owner` определяется просто: если `T` является указателем, то `gsl::owner<T>` становится псевдонимом `T`, иначе определение аннулируется.

Цель этого типа не в том, чтобы явно передать право владения, а в том, чтобы намекнуть пользователю о смене владельца. Вместо внедрения этой информации в имя функции она встраивается в тип. Вполне возможно создать тип с именем `owner`, который выполняет все необходимые действия для правильного отслеживания и передачи права владения, но в этом нет необходимости: с такой задачей прекрасно справляются `std::shared_ptr` и `std::unique_ptr`. Тип `gsl::owner` — это просто синтаксический сахар, который можно добавить в существующий код, не оказывая влияния на него, ABI или особенности выполнения, но влияя на удобочитаемость и простоту восприятия, а также на эффективность работы статических анализаторов и обзоров кода.

По мере роста популярности библиотеки GSL можно ожидать, что IDE будут распознавать ее типы и предупреждать о злоупотреблениях правами владения с помощью визуальных сигналов в редакторе, таких как подчеркивание красной линией или всплывающие подсказки в виде лампочки. Но до тех пор тип `gsl::owner` следует использовать не как средство принудительной передачи права владения, а как описательный тип. В конце концов, относитесь к `gsl::owner` как к последнему средству, когда нет возможности использовать абстракции владения более высокого уровня.

ПОДВЕДЕМ ИТОГ

- Владеть чем-то означает нести ответственность за что-то.
- В C++ имеются интеллектуальные указатели, однозначно определяющие принадлежность и владение.
- Для обозначения владения используйте интеллектуальные указатели или `gsl::owner<T>`.
- Не принимайте на себя права владения, получая простой указатель или ссылку.

ГЛАВА 3.2

I.3. Избегайте синглтонов

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ – ЭТО ПЛОХО

«Глобальные объекты — это плохо. Понятно?» Вы будете постоянно слышать эту фразу и от начинающих, и от опытных программистов. Но давайте разберемся, почему это плохо.

Глобальный объект находится в глобальном пространстве имен. Существует только одно такое пространство, отсюда и название «глобальное». Глобальное пространство имен — это самая внешняя декларативная область единицы трансляции. Имена в глобальном пространстве имен называются глобальными именами. Любой объект с глобальным именем является глобальным объектом.

Такой объект не всегда видим во всех единицах трансляции программы. Правило единственного определения означает, что он может быть определен только в одной единице трансляции, но объявление может повторяться в любом их количестве.

Глобальные объекты не имеют ограничений доступа. Если они видимы, то вы можете взаимодействовать с ними. У глобальных объектов нет иного владельца, кроме самой программы, то есть за них не отвечает ни один другой объект. Глобальные объекты имеют статический класс хранения, поэтому они инициализируются при запуске (на этапе статической инициализации) и уничтожаются при завершении работы (на этапе статической deinициализации).

Это порождает проблемы. Владение имеет фундаментальное значение для рассуждений об объектах. Поскольку у глобального объекта нет владельца, то как можно рассуждать о его состоянии в любой конкретный момент времени? Вы можете вызывать некоторые функции этого объекта, а затем

внезапно, никого не предупредив, другой объект может вызвать другие его функции без вашего ведома.

Хуже того, поскольку глобальные объекты никому не принадлежат, последовательность их создания не определяется стандартом. Вы не сможете с уверенностью сказать, в каком порядке будут создаваться глобальные объекты, а это приводит к довольно неприятной категории ошибок, которые мы рассмотрим ниже.

ШАБЛОН ПРОЕКТИРОВАНИЯ «СИНГЛТОН»

Убедившись во вреде, который наносят глобальные объекты нашему коду, обратим внимание на синглтоны. Впервые члены сообщества C++ столкнулись с этим термином в 1994 году, когда вышла книга *Design Patterns*¹. Она была чрезвычайно захватывающим чтением в то время и остается очень полезной до сих пор. Каждый разработчик должен иметь ее на своей книжной полке или в электронной библиотеке. В ней описываются шаблоны проектирования, повторяющиеся в программной инженерии почти так же, как шаблоны в традиционной архитектуре, такие как купол, портик или галерея. Самое замечательное в этой книге то, что в ней определены общие шаблоны программирования и даны имена. Выбрать хорошее имя — непростая задача, и то, что кто-то взял на себя труд сделать это, стало большим благом.

В книге шаблоны делятся на три основные категории: порождающие, структурные и поведенческие. Именно в категории порождающих шаблонов находится шаблон «Синглтон» (*Singleton*), ограничивающий возможность создания объектов класса единственным экземпляром. Конечно, описание шаблона в такой потрясающей книге подразумевало, что его использование — это хорошо и правильно. В конце концов, мы все использовали синглтоны в течение многих лет, просто не давали им имя, которое было бы принято всеми.

Популярным примером синглтона является главное окно приложения. В главном окне происходят все действия, прием вводимой пользователем

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns. — Reading, MA: Addison-Wesley, 1994 (Гамма Э., Влиссидес Дж., Хелм Р., Джонсон Р. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — Питер, 2016).

информации и отображение результатов работы. В программе может существовать только одно главное окно, поэтому имеет смысл предотвратить создание другого такого окна. Другой пример синглтона — класс менеджера. Подобные классы характеризуются включением в идентификатор слова `manager`. Это явный признак, что на самом деле создан синглтон и имеются проблемы с определением прав владения тем, чем он управляет.

ФИАСКО ПОРЯДКА СТАТИЧЕСКОЙ ИНИЦИАЛИЗАЦИИ

Синглтоны подвержены проблеме фиаско порядка статической инициализации¹. Этот термин был введен Маршаллом Клейном (Marshall Cline) в его сборнике вопросов и ответов по C++ и характеризует проблему создания зависимых объектов не по порядку. Рассмотрим два глобальных объекта, A и B, где конструктор B использует некоторые функции, предоставляемые объектом A, поэтому A должен быть создан первым. Во время компоновки редактор связей идентифицирует набор объектов со статическим классом хранения, выделяет область памяти для них и создает список конструкторов, которые должны быть вызваны до вызова `main`. Вызов этих конструкторов во время выполнения называется статической инициализацией.

Можно определить, что B зависит от A, и поэтому A должен быть создан первым, но нет стандартного способа сообщить компоновщику об этом. Можно ли что-то предпринять? В таком случае нужно найти способ обозначить зависимость в единице трансляции. Но компилятор знает только о той единице трансляции, которую он компилирует.

Мы уже видим, как вы хмурите брови: «А если я скажу компоновщику, в каком порядке их создавать? Можно ли изменить компоновщик, чтобы он соответствовал этой потребности?» На самом деле такая попытка уже была предпринята. Давным-давно используется IDE под названием `Code Warrior` от компании Metrowerks. Версия, которой пользовался я (Гай

¹ Возможно, слово «фиаско» является несправедливой характеристикой. Статическая инициализация никогда и не должна была поддерживать топологический порядок инициализации. Это невозможно при раздельной компиляции, инкрементном связывании и с компоновщиками 1980-х годов. C++ пришлось смириться с существующими операционными системами. Это было время, когда системные программисты еще только учились пользоваться острыми инструментами.

Дэвидсон. — *Примеч. ред.*), предлагала свойство, позволявшее программисту диктовать порядок создания статических объектов. И все было хорошо, пока я случайно не создал малозаметную циклическую зависимость, на трассировку которой уходило почти 20 часов.

Вы можете возразить: «Циклические зависимости — неизбежный спутник разработки. Факт их получения из-за неправильного определения отношений не должен исключать возможности диктовать порядок создания объектов на этапе статической инициализации». Все верно, та проблема была решена, и я продолжил работу. Но не забывайте, что, если понадобится перенести код на другой набор инструментов, не поддерживавший такой возможности, код потеряет работоспособность. Программист может дорого заплатить, если попытается, используя такие конструкции, сделать свой код переносимым.

«Тем не менее, — можете продолжить вы, — эту возможность комитет *мог бы* стандартизировать. Спецификации компоновки уже включены в стандарт. Почему бы не добавить возможность определения порядка инициализации?» Что ж, признаемся: есть еще одна проблема со статическим порядком инициализации. Она заключается в том, что ничто не мешает вам запустить несколько потоков выполнения во время статической инициализации и обратиться к объекту до его создания. А уж при этом точно очень легко выстрелить себе в ногу из-за зависимостей между глобальными статическими объектами.

Комитет не имеет привычки стандартизировать потенциальные мины замедленного действия. Зависимость от порядка инициализации чревата опасностями, как было показано в предыдущих абзацах, и позволять программистам управлять этой возможностью как минимум неразумно. Кроме того, такая стратегия противоречит самой концепции модульной организации. Статический порядок инициализации задается для каждой единицы трансляции в порядке объявления. Задание порядка несколькими единицами трансляции сразу — вот где все рушится. Определяя зависимые объекты в одной единице трансляции, вы избегаете всех этих проблем, сохранив при этом ясность цели и разделение задач.

Слово «компоновщик» (*linker*) встречается в стандарте *только один раз*¹. Компоновщики не уникальны для C++; они связывают любой объектный код, который имеет соответствующий формат, независимо от того, какой компилятор его генерировал, будь то C, C++, Pascal или другие

¹ <https://eel.is/c++draft/lex.name>

языки. Требовать, чтобы компоновщики внезапно начали поддерживать новую возможность исключительно для продвижения рискованной практики программирования на одном языке, — это слишком. Выкиньте из головы идею стандартизации порядка инициализации. Это глупая затея.

Определяя зависимые объекты в одной единице трансляции, вы избегаете всех этих проблем, сохраняя при этом ясность цели и разделение задач.

Теперь, после всего сказанного, рассмотрим способ обойти фиаско порядка статической инициализации. А выход в том, чтобы вывести объекты из глобальной области видимости и тем самым дать возможность запланировать их инициализацию. Самое простое решение — создать функцию, содержащую статический объект требуемого типа, который возвращается функцией по ссылке. Его иногда называют синглтоном Мейерса в честь Скотта Мейерса (Scott Meyers), который описал этот подход в своей книге *Effective C++¹*.

Например:

```
Manager& manager() {
    static Manager m;
    return m;
}
```

Теперь глобальной является функция, а не объект. Объект `Manager` не будет создан до вызова функции: на статические данные в области видимости функции распространяются другие правила инициализации. «Но, — спросите вы, — а как же ситуация конкурентного выполнения? Ведь проблема доступа к объекту из нескольких потоков до его создания никуда не исчезла?»

К счастью, начиная с C++11, это решение стало также потокобезопасным. Если заглянуть в раздел [stmt.dcl]² стандарта, то можно увидеть следующее: «Если поток управления входит в объявление конкурентно, пока инициализация переменной еще не завершилась, то этот поток будет приостановлен до завершения инициализации».

Однако на этом проблемы не заканчиваются: по-прежнему сохраняется риск одновременного обращения к единственному изменяемому объекту без гарантии потокобезопасного доступа к нему.

¹ Meyers S. Effective C++. — Reading, MA: Addison-Wesley, 1998 (Мейерс С. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14).

² <https://eel.is/c++draft/stmt.dcl>

КАК СКРЫТЬ СИНГЛТОН

Взглянув на предложенный выше способ, вы можете решить, что мы просто спрятали синглтон за функцией. Действительно, скрыть синглтон несложно, но в Core Guidelines отмечается, что заставить не использовать его в целом очень трудно. Первая идея выявления синглтонов, предлагаемая рекомендацией «I.3. Избегайте синглтонов», гласит: «Ищите классы с именами, включающими слово `singleton`». Этот совет может показаться вполне действенным, но можно нарваться на другие синглтоны: поскольку синглтон является одним из шаблонов проектирования, инженеры довольно часто добавляют слово `singleton` в имена своих классов, чтобы показать: «Я считаю, что это синглтон» или «Я прочитал книгу *Design Patterns*». Конечно, при этом реализация встраивается в интерфейс, что само по себе очень плохо, но это уже совсем другая история.

Вторая идея, предлагаемая Руководством: «Искать классы, для которых создается только один объект (путем подсчета объектов или изучения конструкторов)». Для этого требуется полный ручной аудит кода по классам. Иногда синглтоны создаются случайно. Можно ввести абстракцию и сформировать из нее класс, а также создать все средства, необходимые для управления жизненным циклом и взаимодействиями с этим классом, такие как специальные функции, общедоступный интерфейс и т. д. Но в конечном счете окажется, что только один экземпляр объекта может существовать в каждый конкретный момент времени. Возможно, в намерения инженера не входило создание синглтона, но именно это и произошло. Подсчет экземпляров показывает, что их количество равно единице.

Последняя идея из Руководства, касающаяся обсуждаемого вопроса: «Если класс X имеет общедоступную статическую функцию, содержащую статическую локальную переменную типа класса X и возвращающую указатель или ссылку на нее, запретите это». Это тот самый метод решения проблемы фиаско порядка статической инициализации, который был описан выше. Класс может иметь надмножество следующего интерфейса:

```
class Manager
{
public:
    static Manager& instance();

private:
    Manager();
};
```

Демаскирующим признаком здесь является приватный конструктор. Объект этого класса может создать только статический член или дружественный класс, но здесь нет объявления дружественных классов. От этого класса нельзя создать производный класс, если не добавить в него другой общедоступный конструктор. Приватный конструктор прямо говорит: «Создание моих экземпляров жестко контролируется другими функциями в моем интерфейсе». И — о чудо! В общедоступном интерфейсе имеется статическая функция, которая возвращает ссылку на экземпляр. Вы, без сомнения, догадаетесь, что именно содержит эта функция-член, взглянув на пример функции `manager()`, приведенный выше.

Вариация этого шаблона — синглтон с подсчетом ссылок. Рассмотрим класс, являющийся жадным пожирателем ресурсов. Из-за этой его особенности желательно не только разрешить существование его единственного экземпляра, но и гарантировать немедленное уничтожение этого экземпляра, как только он станет ненужным. Организовать такое поведение довольно сложно, потому что требуются общий (разделяемый) указатель, мьютекс и счетчик ссылок. Однако вспомните, что это все тот же синглтон, подпадающий под правило «Избегайте синглтонов».

Возможно, сейчас вы смотрите на эту общедоступную статическую функцию-член и говорите себе: «Определенно, в Руководстве должно быть сказано: «Избегайте объектов со статическим классом хранения». В конце концов, это тоже синглтоны». Запомните эту мысль.

ТОЛЬКО ОДИН ИЗ НИХ ДОЛЖЕН СУЩЕСТВОВАТЬ В КАЖДЫЙ МОМЕНТ РАБОТЫ КОДА

При обучении программированию на C++ приводится несколько популярных примеров, описывающих объектно-ориентированные свойства языка. На заправочных станциях, например, есть автомобили, насосы, касса, цистерны для топлива, цены и т. д., это все в совокупности составляет достаточно богатую экосистему для описания многих видов отношений. Точно так же в ресторанах есть столы, клиенты, меню, окно выдачи блюд, официанты, повара, доставщики еды, уборщики и др. (В современных учебниках, вероятно, в качестве подобных моделей могут также упоминаться сайт или аккаунт в Twitter.)

Оба приведенных примера имеют одну общую черту: это абстракция чего-то, существующего в единственном экземпляре. На АЗС имеется одна касса. В ресторане имеется одно окно выдачи блюд. Это точно синглтоны? Если нет, то как тогда быть с созданием объекта?

Одним из возможных разрешений противоречия, которое мы наблюдали, является создание класса с полностью статическим интерфейсом. Все общедоступные функции-члены и приватные данные являются статическими.

Теперь немного отвлечемся и расскажем вам об Уильяме Хите Робинсоне (W. Heath Robinson). Этот английский художник-карикатурист, родившийся в 1872 году в Финсбери-парк в Лондоне, особенно известен своими рисунками нелепо сложных машин, в которых применяется множество ухищрений для решения простых задач. Одна из автоматических аналитических машин, построенных для Блетчли-парк во время Второй мировой войны для помощи в расшифровке немецких сообщений, была названа «Хит Робинсон» в его честь. У него был американский коллега, Руб Голдберг (Rube Goldberg), родившийся в июле 1883 года в Сан-Франциско, который тоже рисовал чересчур сложные устройства и изобрел настольную игру «Мышеловка». Имена этих художников вошли в обиход как синонимы чрезмерной инженерной усложненности.

Примером такой чрезмерной сложности является класс с полностью статическим интерфейсом. Определяя класс, вы создаете общедоступный интерфейс для наблюдения за абстракцией и управления ею, а также множество данных и приватных функций для моделирования поведения абстракции. Однако если данные существуют только в одном экземпляре, то зачем привязывать их к классу? Можно просто реализовать все общедоступные функции-члены в одном исходном файле и поместить данные с приватными функциями в анонимное пространство имен.

Нет, правда, зачем вообще возиться с классом?

Вот оно! Нам пришлось пройти длинный и извилистый путь к этому правильному решению проблемы синглтонов (с маленькой буквы «с»). Они должны быть реализованы как пространства имен, а не классы. Вместо:

```
class Manager
{
public:
    static int blimp_count();
    static void add_more_blimps(int);
    static void destroy_blimp(int);
```

```
private:
    static std::vector<Blimp> blimps;
    static void deploy_blimp();
};
```

вы должны объявить:

```
namespace Manager
{
    int blimp_count();
    void add_more_blimps(int);
    void destroy_blimp(int);
}
```

Реализация не должна быть видна клиенту как какой-нибудь рисунок Хита Робинсона, изумительный и очаровательный в своей сложности. Ее можно спрятать в темных тайниках файла реализации. Такой подход дает дополнительные преимущества: высокую стабильность файла, в котором объявлено пространство имен, и низкую вероятность крупномасштабной зависимой перекомпиляции. Конечно, данные, используемые для моделирования абстракции, не будут принадлежать объекту, а значит, они будут статическими. Поэтому остерегайтесь получить фиаско порядка статической инициализации, как описано выше.

ПОДОЖДИТЕ МИНУТКУ...

Возможно, рассматривая это решение на основе пространства имен, вы замечаете про себя: «Но это все еще «Синглтон»».

Нет, это не «Синглтон». Это синглтон. Проблема, о которой предупреждает Руководство, связана с шаблоном проектирования «Синглтон» (*Singleton*), а не с абстракциями существования чего-то в единственном экземпляре. На самом деле в интервью издательству InformIT в 2009 году Эрих Гамма (Erich Gamma), один из четырех авторов *Design Patterns*, заметил, что у него есть желание удалить шаблон «Синглтон» (*Singleton*) из каталога¹.

Рекомендации, касающиеся языка C++, имеют две проблемы. Первая: данный ранее умный совет не обязательно останется таким же разумным советом с течением времени.

¹ <https://www.informit.com/articles/article.aspx?p=1404056>

На данный момент каждые три года выходит новая редакция стандарта C++. Так, появление `std::unique_ptr` и `std::shared_ptr` в 2011 году изменило ранее звучавший совет о соблюдении парности вызовов `new` и `delete` («Удаляйте объект только в том модуле, в котором он был создан»). Оно сделало возможным отказ от низкоуровневых операций `new` и `delete`, как рекомендуется в «R.11. Избегайте явных вызовов `new` и `delete`». Не всегда бывает достаточно выучить комплекс советов, чтобы затем идти по жизни: по мере развития и изменения языка советы должны постоянно пересматриваться.

Непосредственным проявлением этой проблемы может быть использование привычного фреймворка, предполагающего широкое применение некогда идиоматичных, но ныне устаревших приемов программирования на C++. Возможно, он включает «Синглтон» для захвата и управления переменными среды или настройками, передаваемыми через параметры командной строки, которые могут измениться. Вы можете думать, что ваш любимый фреймворк не ошибется, но это не тот случай. Подобно тому как научное мнение меняется с появлением новой информации, меняются и передовые приемы программирования на C++. Эта книга, которую вы читаете сегодня, может содержать несколько вечных советов, но мы, авторы, считаем, что было бы в высшей степени высокомерно и глупо предполагать, что весь ее текст — это вековечная мудрость с заповедями, высеченными в граните, о том, как следует писать на C++.

Вторая проблема заключается в том, что рекомендации часто являются квинтэссенцией нескольких причин, часто полностью скрытых за выразительной и запоминающейся фразой, закрепляющейся в нашем сознании. Нужно изучить эти причины или хотя бы ознакомиться с ними. «Избегайте синглтонов» гораздо легче запомнить, чем «избегайте чрезмерной разработки абстракций с одним экземпляром в классе и злоупотребления уровнями доступа для предотвращения создания множественных экземпляров». Выучить совет недостаточно. Нужно изучить подоплеку, чтобы знать и понимать, почему предлагается использовать тот или иной подход и когда можно без опаски этого не делать.

Core Guidelines — это живой документ в репозитории GitHub, куда вы можете направлять запросы на извлечение (pull request). В нем содержатся сотни советов, обусловленных различными причинами, и цель этой книги — выделить первопричины возникновения 30 наиболее ценных из них.

Выше мы отмечали, что вы можете подумать, будто все статические объекты являются «Синглтонами» и поэтому следует избегать любых статических объектов. Теперь вы должны понимать, что статические объекты не являются «Синглтонами» и не обязательно являются синглтонами. Они являются экземпляром объекта, продолжительность существования которого совпадает с продолжительностью выполнения программы. И при этом они могут не быть глобальными: область видимости статических переменных-членов ограничена не глобальной областью видимости, а лишь классом.

Данный ранее умный совет не обязательно останется таким же разумным с течением времени.

Точно так же утверждение «Глобальные объекты — это плохо. Понятно?» не всегда верно. Вам может навредить именно изменяемое глобальное состояние, как описывается в рекомендации «I.2. Избегайте неконстантных глобальных переменных». Если глобальный объект неизменяемый, то он является всего лишь свойством программы. Например, разрабатывая физический симулятор для космической игры, мы не без оснований могли бы объявить в глобальном пространстве имен объект типа `float` с именем `G`, представляющий гравитационную постоянную:

```
constexpr float G = 6.674e-11; // Гравитационная постоянная
```

Ведь это универсальная константа, и никто не должен ее менять. Конечно, вы можете решить, что глобальное пространство имен не подходит для таких вещей, и объявить для этих целей пространство имен `universe`:

```
namespace universe {
    constexpr float G = 6.674e-11; // Гравитационная постоянная
}
```

И даже есть вероятность, что вы захотите поэкспериментировать с другой вселенной, а в ней — с другой гравитационной постоянной. В таком случае вы можете использовать функцию, которая просто возвращает значение, а затем изменить логику интерфейса в соответствии с вашей сумасшедшей тягой к экспериментам.

Зная, *почему* глобальные переменные плохи, и помня причины, перечисленные выше, вы сможете решать, когда уместно изменить это правило и все-таки воспользоваться ими, представляя все возможные последствия и принимая всю ответственность за них на себя.

ПОДВЕДЕМ ИТОГ

- Избегайте синглтонов: шаблона, а не абстракции с одним экземпляром.
- Для моделирования абстракции этого типа вместо класса лучше использовать пространство имен.
- С осторожностью используйте статические данные при реализации синглтона.
- Изучайте причины и подоплеку появления рекомендаций в Core Guidelines.
- Пересматривайте советы в Core Guidelines по мере развития языка C++.

ГЛАВА 3.3

С.90. Полагайтесь на конструкторы и операторы присваивания вместо `memset` и `memcp`

В ПОГОНЕ ЗА МАКСИМАЛЬНОЙ ПРОИЗВОДИТЕЛЬНОСТЬЮ

C++ славится своей производительностью, сопоставимой с производительностью «голого железа». Другие языки приходили и уходили, пытаясь оспорить у C++ титул и корону высокопроизводительного языка, но он по-прежнему остается популярным языком для абстракций с нулевыми накладными расходами. C++ унаследовал эту черту от языка C, предлагающего несколько очень эффективных библиотечных функций. Некоторые из них могут быть реализованы как однопроцессорные инструкции.

Рассмотрим для примера функцию `double floor(double arg)`. Она находится в заголовке `<cmath>` и возвращает наибольшее целое значение, не превышающее `arg`. Процессоры x86 имеют единственную инструкцию, способную сделать это, она называется `ROUNDSD`. Интеллектуальный оптимизирующий компилятор может заменить вызов `floor` этой инструкцией и при этом наполнить восторгом душу типичного инженера, жаждущего скорости.

Для этого процессора CISC доступно несколько удивительных инструкций. Если вдруг вам понадобится узнать количество ведущих нулей в значении, чтобы определить ближайшую степень числа 2, для этой цели воспользуйтесь инструкцией `LZCNT`. Или, например, вам потребуется узнать количество установленных битов в значении при вычислении расстояния Хэмминга (Hamming). В этом вам поможет `POPCNT`. Это настолько полезная

инструкция, что Clang и GCC заметят вашу попытку написать ее и заменят ваш код вызовом `POPCNT`. Отличный сервис. Не забудьте оставить чаевые разработчику вашего компилятора.

Когда я впервые начал программировать¹, то быстро перешел с BASIC на язык ассемблера, сначала Z80, затем 68000. Начав изучать C, я смог обращаться с ним как с языком программирования макросов, что и сделало мой переход на C довольно гладким. Я рассуждал о своем коде так, будто он написан на языке ассемблера, только на C его было проще писать, тестировать и отлаживать. Я писал отличный код гораздо быстрее, чем когда использовал ассемблер 68000.

Начиная переходить на C++, я с подозрительностью относился к некоторым его аспектам. Но непродолжительный анализ, опыт и наблюдения обычно развеивали мои подозрения. Например, виртуальные функции выглядели для меня как черная магия, пока я не понял, что это всего лишь указатели на функции, располагающиеся в верхней части моей структуры, хотя они и создавали дополнительный уровень косвенности вдали от ожидаемого места. Перегрузки и шаблоны функций помогли мне избавиться от множества символов и познакомили с возможностью исключения реализации из интерфейса, что позволило писать гораздо более читабельный код.

Но больше всего мне понравился в этом языке синтаксический сахар, позволяющий писать более ясный код. Вещи, замедляющие мою работу, я отбросил: определенно они не нужны мне в путешествии.

Но вот конструкторы... Они оказались худшими из худших.

УЖАСНЫЕ НАКЛАДНЫЕ РАСХОДЫ КОНСТРУКТОРОВ

К моменту, когда я освоил ассемблер, я научился выделять область памяти для работы и заполнять ее нулями одной ассемблерной инструкцией. Если я чувствовал себя более уверенно, то даже не обнулял память, а просто инициализировал ее значениями в соответствии с контекстом, хотя это усложняло отладку, потому что было трудно выяснить, какие ячейки я уже инициализировал, а какие — еще нет.

¹ Экскурс в историю с Гаем Дэвидсоном. Языковые экзерсисы.

Перейдя на С, я быстро научился объявлять целые числа, числа с плавающей точкой и структуры в начале функции, а в отладочных сборках вызывать библиотечную функцию `memset`, объявленную в `<string.h>`, для заполнения памяти нулями одним вызовом. Я просто увеличивал (или уменьшал) указатель стека и заполнял пространство нулями.

После перехода на С++ я был вынужден отучаться от этой привычки и привыкать к конструкторам по умолчанию. Мне пришлось усвоить, что эти конструкторы вызываются несмотря ни на что и нет никакой возможности предотвратить эти вызовы. Я был вынужден смотреть на ассемблерный код и вздрагивать. Ничто не могло сравниться в скорости со «старым добрым способом». Лучшее решение, которое я смог придумать, — вызвать `memset` в теле конструктора. Списки инициализации просто не могли обнулить все одной ассемблерной инструкцией.

Можете представить, как я относился к операторам присваивания и конструкторам копирования. Почему они не вызывали `memset`? Почему не было выбрано более деликатное, изящное и утонченное поэлементное действие? Я мог бы понять такую избирательность, когда действительно нужно что-то сделать в теле конструктора, но зачем такие большие накладные расходы, если я просто выделял область памяти?

Я боролся с языком, проклиная эту неэффективность, получаемую в обмен на более понятный код. Чтобы написать код, для которого производительность была критически важной, иногда я переходил на С, используя тот факт, что оба языка были понятны компоновщику.

Химера — мифический огнедышащий зверь с головой льва, телом козла и хвостом дракона. Я писал эти жуткие химеры в 1990-х. Мне потребовалось много времени, чтобы понять, что главная моя ошибка заключается в слишком раннем объявлении объектов в функциях. Кроме того, это было до стандартизации и введения правила «как если бы». Еще больше времени мне потребовалось, чтобы осознать истинную его ценность. Жизненно важно рассмотреть подробнее правила конструирования объектов.

Стандарт описывает инициализацию классов на 12 страницах, начиная с [class.init]¹ и ссылаясь еще на восемь страниц в [dcl.init]² для случаев, когда нет конструктора. Мы не будем разбирать все эти хитросплетения, а поступим проще и сразу обобщим, начав с агрегатов.

¹ <https://eel.is/c++draft/class.init>

² <https://eel.is/c++draft/dcl.init>

САМЫЙ ПРОСТОЙ КЛАСС

Агрегаты — это классы:

- без объявленных пользователем или унаследованных конструкторов;
- без необщедоступных нестатических переменных-членов;
- без виртуальных функций;
- без необщедоступных или виртуальных базовых классов.

Например:

```
struct Agg {  
    int a;  
    int b;  
    int c;  
};
```

Агрегаты — хорошая штука. Их можно инициализировать, представив значения в фигурных скобках:

```
Agg t = {1, 2, 3};
```

Согласно правилам инициализации, каждый элемент инициализируется копией соответствующего элемента. В примере выше это выглядит так:

```
t.a={1};  
t.b={2};  
t.c={3};
```

Если нет явно инициализированных элементов, каждый элемент инициализируется инициализатором по умолчанию или копией, инициализированной от пустого инициализатора в порядке объявления. Это становится невозможным, если один из членов является ссылкой, потому что ссылки должны связываться при создании экземпляра. Например:

```
auto t = Agg{};
```

При таком объявлении `t` сначала будет инициализирован член `t.a` с помощью {}, затем `t.b` и, наконец, `t.c`. Однако, поскольку все они имеют тип `int`, инициализация превратится в пустую операцию: для встроенных типов нет конструктора. «Ah! — можете воскликнуть вы. Именно здесь я могу вызвать `memset`, это же очевидно. Содержимое структуры не определено, и это плохо, поэтому я могу просто обнулить ее. Это будет совершенно правильно».

Нет. Правильно было бы добавить конструктор, выполняющий эту инициализацию, например, так:

```
struct Agg {
    Agg() : a{0}, b{0}, c{0} {};
    int a;
    int b;
    int c;
};
```

«Но теперь это уже не агрегат, — верно подметите вы, — а мне так нужна эта возможность инициализации фигурными скобками с вызовом `memset`, ну пожалуйста».

Что ж, тогда вы можете использовать инициализаторы членов, например, вот так:

```
struct Agg {
    int a = 0;
    int b = 0;
    int c = 0;
};
```

Если теперь объявить:

```
auto t = Agg{};
```

`t.a` будет инициализирован операцией `= 0`, так же как `t.b` и `t.c`. Более того, можно использовать назначенные инициализаторы (designated initializers), появившиеся в C++20, которые позволяют инициализировать члены объекта разными значениями, например:

```
auto t = Agg{.c = 21};
```

Теперь `t.a` и `t.b` по-прежнему будут инициализироваться `0`, а `t.c` получит значение `21`.

«Да, назначенные инициализаторы хороши, и мы снова получили агрегат» (мы так и чувствуем, как в вашем сознании формируется «но»), «но члены по-прежнему инициализируются по одному! Я хочу использовать `memset` для их инициализации одной инструкцией».

В действительности это плохая идея, потому что инициализация объекта отделяется от его определения. Что получится, если в агрегат будут добавлены новые элементы? Ваш вызов `memset` обнулит только часть агрегата. C++ позволяет заключить весь жизненный цикл объекта в единую

абстракцию — класс, что весьма ценно. Не нужно пытаться идти против течения.

Возможная ваша реплика: «Я буду использовать `sizeof`, чтобы гарантировать независимость от любых изменений в классе».

И все равно это плохая идея. Что, если вы добавите элемент, который по умолчанию инициализируется ненулевым значением? В таком случае вам придется гарантировать, что вызов `memset` учитывает значение этого члена, возможно, путем разделения его на два. Этот несчастный случай затаится и будет ждать момента, чтобы произойти.

«Неубедительно! Я владею агрегатом, он определен в частном файле реализации, а не в заголовке, он *не* будет изменяться без моего ведома, вызов `memset` *совершенно безопасен!* Что за дела? Почему мне нельзя вызвать `memset`?»

В том-то и дело, что на самом деле вызывать `memset` не нужно. Давайте поговорим об абстрактной машине.

О ЧЕМ ГОВОРИТ СТАНДАРТ

«P.2. Придерживайтесь стандарта ISO C++» — одно из первых основных правил в Core Guidelines. Стандарт определяет поведение соответствующей ему реализации C++. Любое отклонение считается нестандартным. Существует множество реализаций C++ для разных платформ, все они действуют по-разному, в зависимости, например, от размера машинного слова и других особенностей, характерных для конкретной цели. Некоторые платформы не имеют энергонезависимого хранилища в виде дисков. Другие не имеют устройства стандартного ввода. Как стандарт учитывает все эти вариации?

Первые три раздела стандарта: «Область применения»¹, «Нормативные ссылки»² и «Термины и определения»³, а также страницы с 10-й по 12-ю четвертого раздела «Общие принципы»⁴ точно объясняют эту проблему.

¹ <https://eel.is/c++draft/intro.scope>

² <https://eel.is/c++draft/intro.refs>

³ <https://eel.is/c++draft/intro.defs>

⁴ <https://eel.is/c++draft/intro>

Вот вам одно из доказательств важности принципа RTFM (Read The Front Matter — «чтение вступительных разделов»).

Первые четыре раздела, составляющие вводную часть, описывают структуру документа, принятые соглашения, значение термина «неопределенное поведение», понятие «плохо сформированная программа» и фактически определяют всю систему отсчета. В разделе «Общие принципы», в частности, в секции [intro.abstract]¹, вы найдете следующий текст: «Семантические описания в этом документе определяют параметризованную недетерминированную абстрактную машину. Этот документ не предъявляет никаких требований к структуре соответствующих реализаций. В частности, они не обязаны копировать или эмулировать структуру абстрактной машины. Скорее, соответствующие реализации должны подражать (и только) наблюдаемому поведению абстрактной машины, как описано ниже».

К этому абзацу прилагается сноска, в которой говорится: «Это положение иногда называют правилом “как если бы”, потому что реализация может игнорировать любое требование этого документа, пока результат получается таким же, как если бы требование было соблюдено, насколько это можно определить по наблюдаемому поведению программы. Например, фактическая реализация не обязана оценивать выражение, если может сделать вывод, что его значение не используется и что при его вычислении не возникают побочные эффекты, влияющие на наблюдаемое поведение программы».

Это изумительная оговорка. Согласно ей, реализация должна лишь эмулировать наблюдаемое поведение. Это означает, что она может просмотреть ваш код, оценить результат его выполнения и сделать все необходимое для соответствия этому результату. Вот как работает оптимизация: берется результат и составляется оптимальный набор инструкций, необходимый для его достижения.

Что это означает для нашего примера агрегатного класса?

Поскольку все инициализаторы членов равны нулю, компилятор заметит, что при создании экземпляра `Agg` требуется обнулить три целочисленных члена. А так как такая инициализация идентична вызову `memset`, он, вполне вероятно, вызовет `memset`. Вызывать `memset` вручную не потребуется.

¹ <https://eel.is/c++draft/intro.abstract>

Но постойте-ка! Класс содержит всего три целых числа. На типичной 64-битной платформе с 32-битными целыми числами это означает, что обнулить нужно только 12 байт. На платформе x64 это можно сделать двумя инструкциями. С чего бы вдруг вы захотели вызвать `memset`? Чтобы убедиться в нашей правоте, посетите веб-сайт Compiler Explorer и попробуйте скомпилировать такой код:

```
struct Agg {  
    int a = 0;  
    int b = 0;  
    int c = 0;  
};  
  
void fn(Agg&);  
  
int main() {  
    auto t = Agg{}; // ❶  
    fn(t);          // ❷  
}
```

Вызов функции в строке ❷ не позволит компилятору оптимизировать объект `t`, отбросив его.

Компилятор gcc для платформы x86-64 с флагом оптимизации `-O3` дает следующий код:

```
main:  
    sub    rsp, 24  
    mov    rdi, rsp  
    mov    QWORD PTR [rsp], 0      // ❶  
    mov    DWORD PTR [rsp+8], 0  
    call   fn(Agg&)            // ❷  
    xor    eax, eax  
    add    rsp, 24  
    ret
```

Как видите, обнуление трех целочисленных членов выполняется двумя инструкциями `mov`. Автор компилятора знает, что это самый быстрый способ обнуления трех целых чисел. Если бы понадобилось обнулить гораздо больше элементов, в игру вступили бы инструкции MMX. Вся прелесть веб-сайта Compiler Explorer в том, что вы легко сможете проверить этот результат сами.

Мы надеемся, что этот пример убедил вас отказаться от использования `memset`.

А КАК ЖЕ MEMCPY?

По аналогии с применением `memset` в программах на С для обнуления структуры логично использовать `memcpу` для присваивания ее другому экземпляру. Присваивание в C++ очень похоже на инициализацию: по умолчанию эта операция копирует данные поэлементно в порядке объявления, используя операции присваивания, соответствующие типам членов. Вы можете написать и свой оператор присваивания, в отличие от конструктора не начинающийся с неявного поэлементного копирования. Можно подумать, что в этой ситуации аргументы в пользу вызова `memcpу` стали сильнее. Но на самом деле по тем же причинам, что были описаны выше, это по-прежнему не является ни хорошей идеей, ни необходимости. Вернемся на сайт Compiler Explorer и внесем небольшое изменение в исходный код:

```
struct Agg {
    int a = 0;
    int b = 0;
    int c = 0;
};

void fn(Agg&);

int main() {
    auto t = Agg{}; // ①
    fn(t);          // ②
    auto u = Agg{}; // ③
    fn(u);          // ④
    t = u;           // ⑤
    fn(t);          // ⑥
}
```

На этот раз компилятор сгенерировал следующий код:

```
main:
    sub    rsp, 40
    mov    rdi, rsp
    mov    QWORD PTR [rsp], 0          // ①
    mov    DWORD PTR [rsp+8], 0
    call   fn(Agg&)
    lea    rdi, [rsp+16]
    mov    QWORD PTR [rsp+16], 0        // ③
    mov    DWORD PTR [rsp+24], 0
    call   fn(Agg&)                  // ④
```

```

mov    rax, QWORD PTR [rsp+16]    // ⑤
mov    rdi, rsp                  // ⑥
mov    QWORD PTR [rsp], rax      // ⑤
mov    eax, DWORD PTR [rsp+24]
mov    DWORD PTR [rsp+8], eax
call   fn(Agg&)
xor    eax, eax
add    rsp, 40
ret

```

Как видите, компилятор использовал тот же трюк `QWORD/DWORD` и сгенерировал четыре инструкции, напрямую копирующие содержимое памяти из исходного объекта. И снова: зачем вам вызывать `memset`?

Обратите внимание, что если отключить оптимизацию, то сгенерированный код будет вести себя в более точном соответствии со стандартом и в меньшей степени будет следовать правилу «как если бы». Этот код генерируется быстрее и в общем случае проще. Коль скоро вы подумываете о возможности использования `memset` и `memcp`, то мы предполагаем, что вас как раз больше заботит оптимизация. В таком случае вам будет достаточно только сгенерировать более оптимизированный код.

В приведенном выше ассемблерном коде можно заметить неожиданное изменение порядка. Автор компилятора знает все о характеристиках выполнения этих инструкций и соответствующим образом переупорядочил код, потому что от него требуется только эмулировать наблюдаемое поведение.

НИКОГДА НЕ ПОЗВОЛЯЙТЕ СЕБЕ НЕДООЦЕНИВАТЬ КОМПИЛЯТОР

Чтобы получить максимальную отдачу от компилятора, нужно сообщить ему, чего именно вы добиваетесь на самом высоком доступном уровне абстракции. Как мы видели, для `memset` и `memcp` доступны более высокие уровни абстракции: конструирование и присваивание.

В качестве последнего примера рассмотрим `std::fill`. Вместо заполнения блока памяти одним значением или копирования объекта, состоящего из нескольких слов, в один блок памяти инструкция `std::fill` решает задачу дублирования объекта из нескольких слов в блок памяти.

Простейшей ее реализацией был бы простой цикл и итеративное конструирование на месте или присваивание существующему объекту:

```
#include <array>

struct Agg {
    int a = 0;
    int b = 0;
    int c = 0;
};

std::array<Agg, 100> a;

void fn(Agg&);

int main() {
    auto t = Agg{};
    fn(t);
    for (int i = 0; i < 1000; ++i) { // Заполнить массив
        a[i] = t;
    }
}
```

Однако все это может сделать `std::fill`, вам придется читать меньше кода, и вы с меньшей вероятностью допустите ошибку, как это произошло выше. (Заметили? Сравните размер массива и количество итераций в цикле `for`.)

```
int main() {
    auto t = Agg{};
    fn(t);
    std::fill(std::begin(a), std::end(a), t); // Заполнить массив
}
```

Разработчики компиляторов прилагают все силы, чтобы сгенерировать наилучший код. Типичная реализация `std::fill` будет включать в себя механизм SFINAЕ (или, что более вероятно в настоящее время, ограничение предложением `requires`), чтобы использовать простую функцию `memcp` для тривиально конструируемых и тривиально копируемых типов, где безопасно можно использовать `memset` и вызов конструктора не требуется.

Цель этой рекомендации не только в том, чтобы отговорить вас от использования `memset` и `memcp`, а также и убедить использовать возможности языка, чтобы дать компилятору всю информацию и тот мог сгенерировать оптимальный код.

Чтобы получить максимальную отдачу от компилятора, нужно сообщить ему, чего именно вы добываетесь на самом высоком доступном уровне абстракции.

Не заставляйте компилятор гадать. Он спрашивает вас: «Что вы хотите, чтобы я сделал?» — и действительно постарается сделать все возможное, получив от вас правильный и максимально полный ответ.

ПОДВЕДЕМ ИТОГ

- Используйте конструкторы и присваивание, а не `memset` и `memcpy`.
- Используйте максимально высокий из доступных уровеней абстракции для связи с компилятором.
- Помогите компилятору сделать для вас все возможное.

ГЛАВА 3.4

ES.50. Не приводите переменные с квалификатором `const` к неконстантному типу

Когда меня¹ спрашивают о моих любимых особенностях C++, помимо детерминированного уничтожения, среди прочих я называю квалификатор `const`. Он позволяет разделить интерфейс между представлением и управлением и дать пользователям возможность изучить способы работы с ним. Это разделение не раз выручало меня, но никогда так сильно, как в моем первом крупном игровом проекте.

На рубеже веков мои работодатели затеяли грандиозный проект — компьютерную игру под названием *Rome: Total War*. Это была стратегия реального времени, действия в которой происходили в эпоху Римской империи. В игре были представлены сражения с сотнями солдат, действующих в составе кавалерийских, пехотных и артиллерийских частей, и предоставлялись широкие возможности управления для перемещения войск по полю боя. Игрок мог объединять отдельные части в формирования, создавать группы из отдельных солдат, отправлять их в бой против вражеских сил, контролируемых хитрым ИИ, и наблюдать, как разворачивается драма в прекрасно оформленном трехмерном мире.

Работы по созданию игры оказалось довольно много, даже больше, чем мы ожидали. Одной из причин этого было желание сделать именно много-пользовательскую игру. Мы хотели дать возможность сражаться не только с искусственным интеллектом компьютера, но также с другими игроками-

¹ Экскурс в историю. Говорит автор Дж. Гай Дэвидсон, и говорит на протяжении целой главы.

людьми. Это породило множество проблем, не последней из которых была поддержка одинакового состояния мира на машинах разных игроков.

Подобная проблема характерна для всех многопользовательских игр. Если вы играете в автогонки, то важно, чтобы на компьютерах всех игроков все автомобили располагались одинаково и двигались синхронно. Если складывается такая ситуация, что на разных игровых компьютерах первыми пересечь финишную черту могут разные автомобили, то такая игра никуда не годится. Одно из возможных решений — назначить одну игровую станцию авторитетным сервером игрового мира, гарантировать регулярную отправку всеми клиентскими компьютерами обновлений их экземпляров игры на сервер, заставить сервер преобразовывать эти обновления в новое состояние мира и рассыпать новое состояние клиентам.

В случае с автогонками это будет означать отправку на клиентские машины, скажем, 20 новых положений и ускорений автомобилей. Нагрузка небольшая: нужны только компоненты x и y (если на гоночной трассе нет мостов), то есть по четыре 4-байтных компонента с плавающей точкой для 20 автомобилей, всего получается 320 байт на обновление мира.

В то время частота обновления 30 кадров в секунду считалась вполне приемлемой, но при этом для каждого кадра требовалось обновлять не весь мир, а только его представление. Модель мира включает координаты и ускорения каждого автомобиля, поэтому с помощью уравнений Ньютона можно точно оценить, где будет находиться каждый из них. Нужно лишь, чтобы обновления мира выполнялись быстрее, чем может распознать человеческий мозг. Десяти герц для этого вполне достаточно, то есть каждую секунду сервер должен отправлять 3200 байт каждому клиенту. Обеспечить такую скорость в последнее десятилетие 1990-х не было проблемой. У всех были модемы на 56 Кбит/с, так что требуемая пропускная способность 26 Кбит/с выглядела приемлемой.

РАБОТА С БОЛЬШИМ КОЛИЧЕСТВОМ ДАННЫХ

К несчастью, такой подход был неосуществим в нашей игре. Автомобиль выполняет одно действие — движется. Солдаты в игре выполняют самые разные действия. Ходят, бегают трусцой, бегают в полную силу, метают копья, размахивают мечами и буксируют осадные машины к городским стенам — это лишь небольшой перечень действий, которые они могут выполнять, причем каждый в отдельности со своей уникальной анимацией.

Это означало, что каждый солдат характеризовался шестью компонентами, а не четырьмя, потому что требовалось включить не только выполняемые ими действия, но и как далеко они продвинулись в этих действиях. Получилось по 18 байт на солдата.

Хуже того, солдат было больше 20. На самом деле, чтобы придать осмысленность игре, нужно было разместить 1000 солдат. Получалась удручающая арифметика:

```
10 Герц *
1,000 солдат *
18 байт *
8 бит =
1,440,000 бит в секунду
```

В начале этого века такая пропускная способность была невозможной. В домохозяйствах только начали появляться сети ADSL, и пропускная способность исходящего канала выше 1 Мбит/с была большой редкостью. Оставалось единственное решение — отправлять каждому клиенту список команд для применения к солдатам и гарантировать их выполнение каждым клиентом. Поскольку команды отдавались только отрядам из нескольких солдат раз в несколько секунд, передавать такие обновления было проще. Конечно, при этом требовалось, чтобы каждый клиент поддерживал идентичную копию игрового мира, известную как синхронное состояние.

Задача была чрезвычайно сложной. Чтобы добиться максимальной определенности, все объекты в игре должны были инициализироваться одинаково. Если в структуру добавлялся новый элемент данных, его нужно было инициализировать строго определенным значением. И даже случайные числа должны были быть определенными. Мы ничего не должны были упустить из виду. Нам пришлось решать очень необычные задачи, например изменять режимы арифметики с плавающей точкой в графических драйверах, что приводило к разным результатам одних и тех же вычислений на разных машинах. Но самая большая проблема заключалась в предотвращении рассинхронизации между представлением и моделью мира.

Каждый клиент имеет свое окно на мир. Предполагалось, что механизм визуализации будет смотреть на мир и вызывать константные функции-члены каждого объекта, чтобы получить необходимую информацию. Использование константных функций давало уверенность, что механизм визуализации не будет мешать механизму моделирования мира.

К сожалению, константные функции работают только до определенного предела. Взгляните на следующий класс:

```
class unit {  
public:  
    animation* current_animation() const;  
  
private:  
    animation* m_animation;  
};
```

Механизм визуализации может удерживать объект `unit const*`, вызвать функцию `current_animation()` и, если получен объект `animation`, внести в него изменения. Объект `unit` (представляющий отряд солдат) не обязательно должен иметь свою анимацию: иногда все солдаты в отряде будут иметь общую анимацию, иногда у каждого может быть своя анимация, например, когда солдат поражается копьем во время марша в строю. Константная функция-член возвращает константный объект указателя по значению, а не константный объект `animation` по указателю.

БРАНДМАУЭР CONST

Эта проблема имеет несколько решений, например, можно реализовать пару функций, в которой константная функция возвращает `animation const*`, а неконстантная — `animation*`, но дело в том, что злоупотребление квалификатором `const` может незаметно приводить к катастрофическим последствиям. Одно маленькое изменение может вдруг просочиться в остальной мир и оставаться незамеченным, пока не станет слишком поздно, чтобы исправить его. Это как эффект бабочки, только более ярко выраженный.

Интерфейс `const`, или, как мы его называли, брандмауэр `const`, был в высшей степени важен. В коде этот квалификатор нес большую уникальную нагрузку: подсказывал, какие функции являются частью представления, а какие — частью контроллера. Злоупотребление брандмауэром `const` оказалось большой медвежью услугу остальным членам команды. По мере продвижения проекта вперед требовалось все больше и больше времени для выяснения причин рассинхронизации мира.

Как вы понимаете, появление `const_cast` в любом месте в коде сродни тревожному звоночку. Налицо был соблазн разрешить различные причудливые

взаимодействия между объектами с помощью `const_cast`, и программистов приходилось удерживать от этого, постоянно напоминая об ужасной судьбе, ожидающей их. Объявление чего-то незыблемым, а затем его изменение — это самый худший обман ваших клиентов.

Например, представьте такую функцию:

```
float distance_from_primary_marker(soldier const& s);
```

Все что угодно должно иметь возможность безопасно вызывать эту функцию для каждого солдата, не опасаясь вмешаться в модель мира. Согласно объявлению, эта функция вообще не меняет состояние солдата. А теперь вообразите, что где-то в теле функции имеется следующий код:

```
float distance_from_primary_marker(soldier const& s) {
    ...
    const_cast<soldier&>(s).snap_to_meter_grid(); // О боже...
    ...
}
```

Производительность вычислений повышается при работе в метровом масштабе, но вместо кэширования текущего значения и выполнения необходимых арифметических действий автор просто переместил солдата на несколько сантиметров, возможно собираясь восстановить его место-положение позже.

Решение, чреватое весьма пагубными последствиями.

РЕАЛИЗАЦИЯ ДВОЙНОГО ИНТЕРФЕЙСА

Повсюду оказался разбросан код с двойным интерфейсом, причем один из интерфейсов был константным, а другой — неконстантным. Функции-члены дублировались уже с квалификатором `const`, например:

```
class soldier {
public:
    commander& get_commander();
    commander const& get_commander() const;
};
```

У каждого солдата есть командир, но, чтобы отыскать его, требуется проверить немало объектов и выполнить несколько запросов. Чтобы

не дублировать код и не увеличивать бремя сопровождения, можно использовать перегруженную версию с квалификатором `const`, вот так:

```
commander& soldier::get_commander() {
    return const_cast<commander&>(
        static_cast<soldier const&>(*this).get_commander());
}
```

Здесь `const_cast` применяется к возвращаемому типу, поэтому разумно предположить, что, поскольку это неконстантная функция, в таком преобразовании нет ничего опасного. Однако это противоречит духу Руководства. К счастью, с тех пор, как в C++11 появились завершающие типы возвращаемого значения, имеется лучшее решение:

```
class soldier {
public:
    commander& get_commander();
    commander const& get_commander() const;

private:
    template <class T>
    static auto get_commander_impl(T& t)
        -> decltype(t.get_commander) {
            // реализация функции
    }
};
```

Общедоступные функции `get_commander` просто перенаправляют вызов шаблону статической функции. Преимущество этого решения в том, что шаблон функции знает, когда он работает с объектом `soldier const`. Квалификатор `const` является частью типа `T`. Если реализация нарушит `const`, то компилятор сообщит об ошибке. Никакого приведения не требуется, и это хорошо, так как приведение смотрится уродливо.

Однако это решение не всегда пригодно. Рассмотрим пример с `current_animation`:

```
class unit {
public:
    animation* current_animation();
    animation const* current_animation() const;

private:
    animation* m_animation;
};
```

Взглянув на этот код, можно подумать, что функция возвращает указатель на объект анимации и только. К сожалению, не все так просто и нужно реализовать много логики, связанной с обработкой этой анимации.

Заманчиво было бы проделать тот же трюк и для преобразований. Однако если вызывающий код должен изменить анимацию, а функция-член с квалификатором `const` не предполагает изменения возвращаемого значения, которым она не владеет, то возникнет ошибка. Впрочем, в этом случае можно с уверенностью предположить, что функции будут иметь разную реализацию.

КЭШИРОВАНИЕ И ОТЛОЖЕННЫЕ ВЫЧИСЛЕНИЯ

Другой пример, который может сподвигнуть на использование ключевого слова `const_cast`, — кэширование результатов дорогостоящих вычислений. Их много в области моделирования мира, даже в подмножестве размером 2 на 2 километра. Как мы уже отмечали, выяснение личности командира — нетривиальная задача, поэтому рассмотрим попытку сохранить значение для ускорения его получения в будущем.

```
class soldier {
public:
    commander& get_commander();
    commander const& get_commander() const;

private:
    commander* m_commander;

    template <class T>
    static auto get_commander_impl(T& t)
        -> decltype(t.get_commander);
};
```

Правила таковы, что если командир погиб, то назначается новый командир. Функция `get_commander_impl` вызывается много раз, тогда как командир, скорее всего, погибнет лишь несколько раз за всю игру, поэтому хранение лишнего указателя в обмен на множество одинаковых вычислений — хороший вариант.

Первое, что сделает функция, — проверит, жив ли текущий командир. Если жив, то она сможет быстро вернуть кэшированное значение. Если нет, то функция продержится через дебри объектов, представляющих мир, выяснит,

кто является командиром, и запишет значение обратно в член `m_commander`, прежде чем разыменовать указатель и вернуть ссылку. Это удивительно дорогостоящая процедура. Мы намеренно использовали словосочетание «продерется через дебри»: с таким количеством объектов, составляющих мир, и с таким количеством различных отношений, которые нужно поддерживать и контролировать, поиск командира можно сравнить с ситуацией, когда вы продираетесь через праздничную толпу на улице, выкрикивая имя своего друга в надежде быстро отыскать его. Так что слежение за командиром — идеальный кандидат для кэширования.

Беда в том, что этот шаблон функции должен работать как с объектами типа `soldier const`, так и `soldier` — в первом случае изменение указателя будет запрещено. Единственное решение — `const_cast t`:

```
template <class T>
auto soldier::get_commander_impl(T& t) -> decltype(t.get_commander) {
    if (!t.m_commander->is_alive()) {
        ... // Поиск нового командира
        const_cast<soldier&>(t).m_commander = new_commander;
    }
    return *t.m_commander;
}
```

Очень некрасиво.

ДВА ВИДА CONST

Есть две интерпретации «константности». Преимущество, которое дает `const`, заключается в отсутствии заметной разницы в природе объекта, возвращаемого вызовом константной функции-члена. Предположим, что объект является полностью автономным и не ссылается на другие объекты, подобно чистой функции, которая не ссылается на данные, внешние по отношению к ее области видимости. В этом случае последовательные вызовы константных функций-членов всегда будут давать одни и те же результаты.

Это не означает, что представление объекта останется неизменным. Такой уровень абстракции был бы неверным. Реализация не ваша забота сейчас, видимый интерфейс — вот что должно вас волновать.

На самом деле рассматриваемая сейчас нами проблема затрагивает важный аспект проектирования классов. Чем владеет ваш класс, с чем

разделяет права владения и в чем просто заинтересован? В примере с классом `unit` некоторые переменные-члены хранят указатели на другие объекты, принадлежащие кому-то другому. Что означает квалификатор `const` в этом случае?

В этом случае он может означать лишь, что функция не изменяет связанный с ней объект каким-либо наблюдаемым образом. Такая константность известна как логическая.

Другой вид константности известен как побитовая константность. Она налагается автором функции-члена с квалификатором `const` и означает, что никакая часть представления объекта не может быть изменена во время выполнения функции. Это требование гарантируется компилятором.

Присутствие `const_cast` внутри функции-члена с квалификатором `const` должно вызвать у вас муки совести. Вы обманываете своих клиентов.

К сожалению, пример с кэшированием не единственное место, где может понадобиться использовать `const_cast`. Представьте, что вы решили гарантировать потокобезопасность вашего класса. Одно из возможных решений — добавить переменную-член с мьютексом и блокировать его при выполнении функций-членов. Мы не можем оставить это последнее предложение без внимания и хотели бы отметить, что вы должны свести к минимуму досягаемость мьютекса: он должен охватывать как можно меньше данных и быть частью наименьшей возможной абстракции. Это перекликается с рекомендациями «ES.5. Минимизируйте области видимости» и «CP.43. Минимизируйте время выполнения критической секции».

Но использование мьютекса-члена влечет за собой еще одну проблему: как заблокировать его в функции-члене с квалификатором `const`, если для этого требуется изменить состояние мьютекса?

Теперь, определив разницу между логической и побитовой константностью, можно ввести ключевое слово `mutable`. Цель этого ключевого слова — учитывать логическую константность объекта, не учитывая побитовую константность. Это давнишнее ключевое слово, его можно найти в довольно старых программных проектах. Оно декорирует переменные-члены, показывая, что они могут изменяться во время выполнения константных функций-членов.

Но вернемся к примеру с классом `unit` и посмотрим, как можно его использовать:

```

class soldier {
public:
    commander& get_commander();
    commander const& get_commander() const;

private:
    mutable commander* m_commander; // Иммунитет от ограничений const

    template <class T>
    static auto get_commander_impl(T& t)
        -> decltype(t.get_commander) {
        if (!t.m_commander->is_alive()) {
            ... // Поиск нового командира
            t.m_commander = new_commander; // Всегда доступно для изменения
        }
        return *t.m_commander;
    };
};

```

Теперь член `m_commander` можно изменять даже по константной ссылке.

СЮРПРИЗЫ CONST

Очевидно, что не следует произвольно разбрасывать `mutable` по всему классу. Это ключевое слово должно применяться только к данным, используемым для подсчетов, а не для моделирования абстракции. В отношении члена мьютекса это разумная политика. Однако в примере выше это не так очевидно. Простые указатели сбивают с толку и мутят воду в процессе определения владения, как обсуждалось в рекомендации «I.11. Никогда не передавайте владение через необработанный (простой) указатель (`T*`) или ссылку (`T&`)». Они также вносят путаницу в дизайн API, когда дело доходит до отделения представления от контроллера.

Например, следующий фрагмент кода является вполне законным:

```

class int_holder {
public:
    void increment() const { ++ *m_i; }

private:
    std::unique_ptr<int> m_i;
};

```

Он вызовет удивление у типичного пользователя, предполагающего невозможность изменения объекта `std::unique_ptr` и его содержимого.

На самом деле `const` распространяется только до объекта `std::unique_ptr`. Разыменование объекта — это константная операция, потому что она не изменяет `std::unique_ptr`. Это подчеркивает важность понимания разницы между константным указателем и указателем на константу. Константный указатель нельзя изменить, но объект, на который он указывает, можно. Это чем-то напоминает поведение ссылок: их нельзя переустановить — они не могут ссылаться на другой объект в течение всего времени своего существования, но объект, на который они ссылаются, изменить можно.

Не следует произвольно разбрасывать `mutable` по всему классу. Это ключевое слово должно применяться только к тем, используемым для подсчетов, а не для моделирования абстракции.

Кстати говоря, ссылки не могут преподнести такой сюрприз:

```
class int_holder {
public:
    void increment() const { ++ m_i; }

private:
    int& m_i;
};
```

Этот код не будет компилироваться. Однако хранить объекты по ссылке в классах редко бывает хорошей идеей: это полностью исключает присваивание значением по умолчанию, потому что ссылку нельзя переустановить.

Что действительно необходимо, так это форма идиомы `rImpl`, объединяющая указатель с объектом внутри одного класса и распространяющая константность на него. Такой объект был предложен для стандартизации и на момент написания книги был доступен в библиотеке `fundamentals v2` под названием `std::experimental::propagate_const`. Ожидается и другие решения, следите за событиями.

ПОДВЕДЕМ ИТОГ

- Ключевое слово `const` чрезвычайно ценно и для дизайна вашего интерфейса, и для ваших клиентов, оно помогает четко разделить API между представлением и управлением.

- Не вводите никого в заблуждение, говоря о константности и используя при этом `const_cast`.
- Осознайте разницу между логической и побитовой константностью.
- При необходимости используйте `mutable`, желательно только для переменных-членов, служащих для подсчетов, а не для моделирования абстракции.
- Помните о том, как далеко распространяется действие `const`, и о разнице между константным указателем и указателем на константу.

ГЛАВА 3.5

Е.28. При обработке ошибок избегайте глобальных состояний (например, errno)

ОБРАБАТЫВАТЬ ОШИБКИ СЛОЖНО

Наверное, где-то есть люди, которые пишут совершенный и безошибочный код. К сожалению, у нас, простых смертных, при разработке кода иногда все идет не по плану. Причины этого могут быть разные: задача была определена неверно, переполнился буфер ввода или сломалось железо. Обычно программы не могут просто метафорически пожать плечами, пробормотать «ну ладно» и прекратить работу или просто попытаться продолжить выполнение в меру своих возможностей. Программы не люди, это машины, не имеющие сознания. Они точно и беспрекословно выполняют все, что им приказывают, хотя иногда в это трудно поверить.

Обработка ошибок имеет долгую, многообразную и пеструю историю. Слово «многообразный» здесь подчеркивает факт существования множества способов обработки ошибок и отсутствие универсального средства.

ЯЗЫК С И ERRNO

Цель этой рекомендации — показать, как избежать проблем, связанных с передачей ошибки через глобальное состояние, поэтому начнем с него. Когда речь заходит о глобальном состоянии, на ум сразу же приходят две проблемы:

- принадлежность к потоку;
- ссылка на ошибку.

Проблема принадлежности к потоку достаточно проста: в современном многопоточном мире глобальный объект ошибки должен содержать информацию о потоке, сообщившем об ошибке. Можно попробовать использовать объект с классом хранения в локальном потоке, но это будет означать, что ошибки, возникшие за пределами конкретного потока, будут скрыты. Другой вариант: создать отдельный объект с информацией об ошибках для каждого потока, что требует тщательной синхронизации. Синхронизацию вполне можно осуществить, но все-таки это требование снижает привлекательность такого подхода для многих разработчиков в их личном рейтинге предпочтительных решений для обработки ошибок.

Конечно, так было не всегда, и унаследованный старый код действительно будет содержать ссылки на объект с именем `errno`. Этот объект — часть стандарта C и имеет простой тип `int`. Начиная с C++11, объект `errno` получил класс хранения в локальном потоке, а до этого он имел статический класс. Это не самое плохое решение: если что-то пойдет не так, появится ошибка, вы наверняка захотите вызвать функцию обработки ошибок, чтобы исправить ее. Одиночный код ошибки означает, что совершенно неважно, где произошла ошибка, функция обработки ошибок сможет определить, что именно пошло не так. Код ошибки можно разделить на группы битов: одни биты определяют область ошибки, другие — ее характер, а третий биты добавляют дополнительный контекст.

Но представьте, что получится, если во время перехода к коду обработки ошибки произойдет другая ошибка. Ваш код обработки может обрабатывать только самую последнюю ошибку, что затрудняет восстановление программы до работоспособного состояния. Кроме того, подход с единственным значением просто плохо масштабируется. Любой код должен проверять состояние после вызова функции в случае, если она сообщила об ошибке. Даже если функция не сообщает об ошибках, это не значит, что она не изменится и не начнет сообщать о них в будущем. Наконец, клиенты могут и будут игнорировать или забывать коды ошибок, независимо от того, насколько настойчиво вы в своей документации советуете им сохранять бдительность и не делать этого.

Вся ситуация здорово напоминает «Глобальные объекты — это плохо. Понятно?». Поскольку объект `errno` не имеет владельца, невозможно контролировать, как им распоряжаются в необычных обстоятельствах. Что будет, если в процессе обработки ошибки произойдет еще одна ошибка? Как отмечает Руководство, глобальным состоянием трудно управлять.

К сожалению, как уже говорилось выше, `errno` является частью стандарта, и некоторые разделы стандартной библиотеки языка С продолжают использовать `errno`, чтобы сообщать об ошибках. Например, если попытаться вычислить квадратный корень из отрицательного числа вызовом `sqrt`, в переменную `errno` будет записано значением `EDOM`.

Проблемы с передачей ошибок через `errno` осознаны давно, еще до появления C++. С тех пор было рассмотрено множество альтернатив, успешных и не очень, просочившихся в программные проекты по всему миру, и часть из них была принята в качестве стандартной практики. Давайте рассмотрим некоторые из подходов к обработке ошибок.

КОДЫ ВОЗВРАТА

Самый распространенный способ сигнализировать об ошибке — просто сообщить о ней вызывающему коду напрямую. Организовать это можно с помощью возвращаемого значения. Если функция имеет ограниченный диапазон значений результата, то значения не из этого диапазона можно использовать, чтобы сигнализировать об ошибке. Например, `scanf` возвращает количество аргументов, которым было успешно присвоено значение, то есть число, которое больше нуля или равное нулю, а об ошибке сигнализирует, возвращая `EOF`, то есть отрицательное значение.

Этот метод имеет очевидные недостатки. Первый: возвращаемый тип должен соответствовать диапазону значений результата и кодов ошибок. Рассмотрим функцию из стандартной библиотеки, вычисляющую натуральный логарифм числа:

```
double log(double);
```

Концептуально область результатов полностью охватывает возвращаемый тип, и нет возможности вернуть признак ошибки. Однако стандарт предусматривает возврат специальных значений, сигнализирующих о том, что что-то пошло не так; для этого используется несколько значений типа `double`, выделенных для представления ошибок. Это может стать неожиданностью для начинающего программиста.

Другой недостаток: вызывающие программы могут игнорировать коды возврата, если только функция не отмечена атрибутом `[[no_discard]]`. Но даже в этом случае вызывающий код может игнорировать объект с кодом

возврата. Хотя в этом случае ответственность незаметно перекладывается на вызывающий код, ведь никто не заинтересован в том, чтобы допускать отбрасывание ошибок.

Другой способ вернуть код ошибки — передать функции ссылку на переменную, через которую можно сообщить об ошибке. Вот как выглядит такой API:

```
double log(double, int&);
```

Вызывающий код предоставляет место для сообщения об ошибках и после вызова проверяет наличие сообщения. В этом решении тоже не все хорошо. Во-первых, оно некрасивое. Читаемые API просто рассказывают о происходящем: если вам нужен логарифм, то вы предполагаете, что есть функция, принимающая и возвращающая число. Присутствие дополнительного параметра создает дополнительную когнитивную нагрузку.

Во-вторых, такое решение несет дополнительные накладные расходы, даже когда вы абсолютно уверены в диапазоне чисел, передаваемых функции. Такой подход часто приводит к API с двойными функциями, где каждая функция имеет соответствующую перегруженную версию, принимающую дополнительный параметр для возврата ошибки. Это по меньшей мере не очень элегантно.

ИСКЛЮЧЕНИЯ

Вместе с C++ появились конструкторы — специальные функции, которые ничего не возвращают и в некоторых случаях не принимают аргументов. Здесь не помогут никакие манипуляции с сигнатурами: вы не сможете сигнализировать об ошибке, не пожертвовав конструктором по умолчанию. Хуже того, все деструкторы имеют одну сигнатуру, не предусматривающую аргументов и ничего не возвращающую. Аналогично ведут себя перегруженные операторы.

Для таких случаев и структур были введены исключения. Они создают дополнительный путь возврата, по которому можно проследовать в обход обычного пути и выполнить все необходимые заключительные операции, обычно производимые в конце функции. Кроме того, исключения можно перехватывать в любом месте в стеке вызовов, а не только в месте вызова функции, что позволяет программисту обрабатывать ошибки там,

где это возможно, или просто игнорировать их, позволяя им подниматься выше по стеку вызовов.

Обработка исключений — дорогое удовольствие. Конец функции очевиден, это оператор `return` или закрывающая фигурная скобка. Исключение может возникнуть во время любого вызова и требует много вспомогательного кода для управления им. Всякий раз, когда вызывается функция, она может сгенерировать исключение, поэтому компилятор должен вставить в это событие весь код, необходимый для очистки стека. Такой дополнительный код не только влияет на скорость выполнения, но и занимает место. Накладные расходы при этой реализации оказываются настолько велики, что все компиляторы предоставляют возможность отключить раскрутку исключений в обмен на невозможность использовать блоки `try/catch`.

Это действительно плохое решение. Результаты опроса, проведенного C++ Foundation в 2019 году¹, показали, что сообщество C++ разделено на две части: около половины всех проектов на C++ полностью или частично запрещают исключения. И еще один важный нюанс: запрет исключений лишает проекты возможности обращаться к частям стандартной библиотеки, использующим исключения для сигнализации об ошибках.

<SYSTEM_ERROR>

Второй стандарт C++, C++11, представил еще одно решение: типы `std::error_code` и `std::error_condition`, которые можно найти в заголовке `<system_error>`. Этот механизм сигнализации об ошибках позволяет стандартизировать отчеты об ошибках, сгенерированных операционной системой или низкоуровневыми интерфейсами в случае автономных систем. Он включает не только код, но и указатель на категорию ошибки. Такое нововведение позволяет программистам создавать и анализировать новые семейства ошибок, производные от базового класса `error_category`.

Упомянутые типы решают проблему подобия ошибок друг другу. Перечисление `errc` определяет большой набор кодов ошибок, импортированных из POSIX, и дает им более понятные имена. Например, `ENOENT` преобразуется в `no_such_file_or_directory`. Предопределенные категории ошибок включают `generic_category`, `system_category`, `iostream_category` и `future_category`.

¹ <https://isocpp.org/files/papers/CppDevSurvey-2019-04-summary.pdf>

К сожалению, подобный прием — всего лишь обновление для `errno`. Это все еще объект, который так или иначе нужно вернуть вызывающему коду. Он не сохраняется в глобальном состоянии и не распространяется по стеку, если не отправить его вместе с исключением. Поэтому даже притом, что это решение дает хороший способ отказаться от использования `errno`, оно не избавляет от многих проблем, свойственных обработке ошибок.

Что ж поделаешь.

BOOST.OUTCOME

Естественно, были и другие попытки исправить ситуацию с обработкой ошибок, и предлагались альтернативные решения. Одно из них можно найти на сайте [boost.org¹](https://www.boost.org/doc/libs/develop/libs/outcome/doc/html/index.html) — в богатом источнике классов, помогающих улучшить код. В Boost есть два класса для решения проблемы обработки ошибок.

Один из них — `result<T, E, NoValuePolicy>`. Первый параметр здесь представляет тип возвращаемого объекта. Второй — тип объекта с информацией о причине сбоя. Экземпляр `result` будет содержать либо экземпляр `T`, либо экземпляр `E`. В этом отношении он очень похож на вариантный тип. О третьем параметре можно прочитать в документации Boost. Хотя, признаемся, для большинства целей достаточно значения по умолчанию, а документация довольно сложная.

Создав объект для возврата чего-то, можно дополнитель но предложить преобразование в значение типа `bool`, сообщающее об успехе вызова или о наличии ошибки, а также функции для выяснения характера ошибки. Поскольку это либо `T`, либо `E`, то в случае нормального выполнения это ничего не будет стоить. Вот как можно использовать этот тип:

```
outcome::result<double> log(double); // прототип функции
```

```
r = log(1.385);
if (r)
{
    // Нормальное продолжение выполнения
}
else
{
    // Обработка ошибки
}
```

¹ <https://www.boost.org/doc/libs/develop/libs/outcome/doc/html/index.html>

Это замечательное решение: оно предлагает согласованный способ проверки наличия ошибок, при этом информация об ошибке в случае ее возникновения помещается в объект. Все локально. Единственный недостаток — можно забыть проверить наличие ошибки.

Второй интересный класс — `outcome<T, EC, EP, NoValuePolicy>`. Если класс `result` сигнализирует об успешном или неуспешном выполнении в сочетании с контекстом, то класс `outcome` представляет ошибки двумя способами: как ожидаемые и как неожиданные, которые задаются вторым и третьим параметрами. Ожидаемую ошибку можно исправить, а неожиданную — нет. Появление неустранимой ошибки — это ситуация, когда обычно принято генерировать исключение.

Этот класс — способ предохранения кода от исключений. Он может передавать исключения между уровнями программы, которые изначально не предусматривали обработку исключений.

Несмотря на удобство, тип `boost::outcome`, являющийся частью библиотеки Boost, не стандартизирован (подробнее об этом — чуть ниже) и поэтому доступен только в коде, где разрешено использовать Boost.

Существует поразительное количество окружений, где ситуация иная, из-за чего остаются доступными только три варианта обработки ошибок: исключения, передача кодов ошибок через стек вызовов или развертывание своих механизмов обработки ошибок, усовершенствованных и приспособленных для нужд предметной области.

Такое положение вещей не кажется удовлетворительным.

ПОЧЕМУ ОБРАБАТЫВАТЬ ОШИБКИ ТАК СЛОЖНО

Первая проблема в том, что ошибки разных типов должны обрабатываться по-разному. Выделим три типа ошибок.

Представьте функцию преобразования строки в число. Преобразование потерпит неудачу, если строка содержит еще что-то, кроме цифр. Это исправимая ошибка, о которой следует сообщить вызывающей стороне: в вызывающем коде имеет место логическая ошибка, из-за чего входные

данные не соответствуют ограничениям, указанным автором функции. То есть здесь самый простой вид ошибок: «Я ожидал это, вы передали мне неправильное значение, поэтому пойдите и подумайте, что вы сделали». Налицо явное нарушение предварительного условия, и вызывающий код всегда должен извещаться о подобных ошибках.

Следующий вид ошибок — ошибка программирования. Здесь неправильное действие совершил не вызывающий, а вызываемый код, например попытавшись разыменовать область памяти, которую не должен был. Беда таких ошибок в том, что они ставят программу в ситуацию, когда та должна остановиться. Из-за ошибки невозможно рассуждать о происходящем, и нет смысла сообщать о ней вызывающему коду. Разве они смогут как-то исправить ситуацию? Программа оказывается в поврежденном состоянии, восстановить которое невозможно.

Теперь об ошибках последнего типа, которые не так однозначны. Они происходят при появлении нарушений в программном окружении. Во вступительной части стандарта вместе с правилом «как если бы» описывается «абстрактная машина» [intro.abstract]¹. Есть несколько способов нарушить работу абстрактной машины, но наиболее часто встречается исчерпание свободного пространства в хранилище или в стеке. В обоих случаях возникает нехватка памяти. Исчерпать свободное пространство в хранилище можно, запросив выделить место для вектора с миллиардом элементов типа `double` на 32-битной машине. Чтобы исчерпать стек, можно запустить бесконечную рекурсию.

Только в первом случае, при ошибке, которая считается исправимой, следует уведомить вызывающий код, сгенерировав исключение или вернув код ошибки. Во всех других случаях нужно просто остановить программу и сообщить об этом программисту через утверждение `assert`, запись в файл журнала или стандартный вывод. Но помните, что в случае нарушения абстрактной машины ваши возможности сообщить об этом программисту могут быть довольно ограниченными.

Вторая проблема в том, что вы не знаете, как вызывающий код среагирует на ошибки. Нельзя полагаться на то, что он обработает ошибку. Он может намеренно отбросить код ошибки и продолжать свою веселую жизнь, а может просто забыть о ней.

¹ <https://eel.is/c++draft/intro.abstract>

Например, функция преобразования строк, упоминавшаяся выше, может возвращать значение типа `double` и вместе со строкой для преобразования принимать ссылку для возврата кода ошибки. Этот код ошибки может быть просто проигнорирован. Проблема игнорирования возвращаемых кодов частично решается добавлением атрибута `[[no_discard]]`, но только если возвращаемое значение используется исключительно для возврата ошибок. И даже в этом случае вызывающий код может принять возвращаемое значение в локальную переменную и забыть об этом. В любом случае заставлять пользователей проверять наличие ошибок — не лучшая идея, ведь тогда решение о том, как поступать в случае сбоя, будет приниматься ими, а не вами.

Вы не знаете, как вызывающий код среагирует на ошибки. Нельзя полагаться на то, что он обработает ошибку.

Проблема в том, что вариантов неправильного выполнения кода намного больше, чем правильного. По аналогии с человеческим телом: в наших силах точно определить, находится ли оно в хорошем рабочем состоянии, но разобраться в бесчисленном множестве причин, по которым его состояние может ухудшиться, — сложная задача.

СВЕТ В КОНЦЕ ТУННЕЛЯ

Мы уже отметили, что решения Boost не стандартизированы и доступны только там, где разрешено использовать эту библиотеку. Комитет уже рассматривает несколько документов с целью улучшить ситуацию и с ошибками, и с Boost.

Первый из них предлагает добавить в стандарт тип `std::expected<T, E>`, который может содержать значение типа `T` или `E`, причем `E` представляет причину отсутствия значения типа `T`. Этот тип больше похож на специализированную версию `std::variant`. Впервые он был предложен вскоре после публикации стандарта C++14, то есть его рассматривают уже довольно долго. Проверить, как движется дело, можно, изучив документ P0323¹.

Второй документ определяет детерминированные исключения с нулевыми издержками, которые генерируют значения, а не типы. Этот документ направлен на объединение разделившегося на почве обработки ошибок

¹ www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0323r10.html

сообщества C++. Он старается сделать обработку исключений намного более приемлемой. В частности, генерируемые исключения размещаются в стеке и типизируются статически, благодаря чему нет необходимости выделять место в куче и не используется механизм RTTI. Детерминированные исключения похожи на особый возвращаемый тип. Узнать, как движется дело с рассмотрением этих предложений, можно, изучив документ P0709¹.

Обратите внимание, что Boost.Outcome является частичной, библиотечной реализацией этой идеи. Для полноценной реализации необходимо внести изменения в язык.

Третий документ предлагает в дополнение к детерминированным исключениям с нулевыми издержками ввести новый объект `status_code` и стандартный объект ошибки, а также некоторые улучшения по сравнению с заголовком `<system_error>`. К ним относится, например, отказ от включения заголовка `<string>`, который подразумевает присутствие большого количества дополнительных механизмов, таких как аллокаторы и алгоритмы. Он значительно увеличивает тем самым время сборки и компоновки.

Еще один из аспектов обработки ошибок, который пока не упоминался, — использование макроса `assert` для обнаружения экземпляров ошибок второго типа, описанных в предыдущем разделе. С его помощью программист задает ожидаемое состояние абстрактной машины в определенный момент, а макрос проверяет фактическое состояние и останавливает программу, если фактическое состояние окажется отличным от заданного. Ошибки этого вида отличаются от ошибок нарушения состояния абстрактной машины: они считаются ошибками программиста.

В развитие этой идеи рассматривалось еще одно предложение, которое чуть не вошло в C++20. Фактически дело дошло до стадии рабочего проекта, но в последнюю минуту продвижение было остановлено. Мы говорим о контрактах. Они определяют предварительные и заключительные условия на уровне языка, а не на уровне библиотечного макроса. Тем самым программист получает возможность снабжать определение функции атрибутами, определяющими также и ожидания.

К сожалению, на момент написания этих строк такое множество разнообразных обновлений языка и библиотеки еще не было включено в рабочий

¹ www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0709r4.pdf

проект стандарта C++. В сообществе пользователей C++ многое остается неизменным на протяжении долгих лет. Программисты все еще используют старые стратегии обработки ошибок в стиле C из-за отсутствия лучшего варианта.

Будущее выглядит светлым, но оно еще не наступило.

ПОДВЕДЕМ ИТОГ

- Глобальное состояние не годится для обработки ошибок. Сохраняйте состояние ошибки локально и всегда обращайте на него внимание.
- Если есть возможность генерировать и перехватывать исключения, используйте их.
- Различайте типы ошибок: неверный ввод, неверная логика, неверное окружение и т. д.
- Если обработка исключений вам недоступна, подумайте о возможности использования Boost.Outcome, но не забывайте мониторить предстоящие изменения в стандарте. Core Guidelines не охватывает этот подход, потому что он выходит за рамки стандарта.

ГЛАВА 3.6

SF.7. Не используйте `using namespace` в глобальной области видимости в заголовочном файле

НЕ ДЕЛАЙТЕ ЭТОГО

Пожалуйста, не делайте этого. Никогда.

Никогда. Ни за что. Ни разу.

Пожалуйста.

В Core Guidelines приводится следующий пример:

```
// bad.h
#include <iostream>
using namespace std; // плохо <- "Ты не шутишь", – сказал Гай

// user.cpp
#include "bad.h"

// некоторая функция с именем copy
bool copy(/*... некоторые параметры ...*/);

int main()
{
    // теперь выбор между локальной версией ::copy и std::copy
    // может быть неоднозначным
    copy(/*...*/);
}
```

«Плохо» — и это еще мягко сказано. Наверняка вам кажется, будто мы мусолим очевидные банальности, но на самом деле этот пример раскрывает только часть ужаса.

НЕОДНОЗНАЧНОСТЬ

Существует одно и только одно глобальное пространство имен. Вот выдержка из стандарта [basic.scope.namespace]¹: «Самая внешняя декларативная область единицы трансляции одновременно является глобальным пространством имен. Имя, объявленное в глобальном пространстве имен, имеет глобальную область видимости. Потенциальная область видимости такого имени начинается в точке его объявления и заканчивается в конце единицы трансляции, являющейся его декларативной областью. Имя с глобальной областью видимости называется глобальным именем».

Не загрязняйте глобальное пространство имен: это ценный ресурс.

Пространства имен — полезный инструмент инкапсуляции, объединяющий связанные символы в одну область и позволяющий повторно использовать имена.

Мой² любимый пример — `vector`. В моей предметной области оно имеет два значения: непрерывный контейнер, способный автоматически изменять свой размер, и кортеж чисел, представляющий математическую величину. Первый находится в пространстве имен `std`, а второй — где-то еще. В моем случае он находится в пространстве имен `gdg::maths`. Пространство имен `gdg` — мое личное, его я использую во всех своих программах, а пространство имен `maths` содержится в нем и объединяет математические типы и функции, такие как `matrix`, `vector`, `normalize`, `intersect` и другие типы, которые могут понадобиться в геометрических вычислениях. Я британец, поэтому использую имя `maths`³. И это не обсуждается.

Наличие двух классов с именем `vector` не является проблемой, потому что для устранения неоднозначности можно уточнить имя префиксом

¹ <https://eel.is/c++draft/basic.scope.namespace>

² Экскурс в историю. Об именовании с Гаем Дэвидсоном.

³ В США и англоговорящих районах Канады предпочли бы имя `math`, без `s` на конце. — *Примеч. пер.*

с названием пространства имен. Тогда `std::vector` будет служить для обозначения типа контейнера в пространстве имен `maths`, а `gdg::maths::vector` — для манипулирования геометрическими объектами за пределами пространства имен `maths`.

Ввод дополнительных символов `std::` для устранения неоднозначности — это невысокая плата. С другой стороны, префикс `gdg::maths::` выглядит несколько длиноватым. Он мешает быстрому восприятию кода. Взгляд постоянно спотыкается о дополнительные двоеточия. К счастью, можно ввести:

```
using gdg::maths::vector;
```

в начале области, и компилятор выберет этот вектор вместо `std::vector` в процессе разрешения символа. Это называется объявлением `using` и, заметьте, отличается от директивы `using`:

```
using namespace gdg::maths;
```

Проблема использования директивы `using` в заголовочном файле в глобальной области видимости очевидна: используя такой прием, вы скрываете от пользователя тот факт, что все символы из пространства имен были внедрены в родительскую область видимости каждой функции и каждого определения класса. Хуже того, если внедрить другую директиву `using` в другой заголовочный файл и подключить оба, то образуется зависимость от порядка их подключения. Можно получить немного отличающийся результат компиляции, если изменить их порядок.

И это, к сожалению, только начало безумия.

ИСПОЛЬЗОВАНИЕ USING

Ключевое слово `using` имеет четыре формы использования. Первая — определение псевдонима. Например:

```
using vector_of_vectors = std::vector<gdg::maths::vector>;
```

Это очень специфическое использование. Оно вводит новое имя — сокращенную форму другого имени. В данном случае вводится имя `vector_of_vectors`, прямо отображающееся в стандартный вектор моих векторов из `maths`. Эта форма использования ключевого слова `using` часто применяется, чтобы сократить объем ввода с клавиатуры и сделать код более ясным.

Вторая форма предназначена для импорта членов класса. Например, так:

```
struct maths_vector : std::variant<vector<float, 2>, vector<int, 2>> {
    using variant::variant;
}
```

Здесь в производный класс вводится член базового класса. Такой прием позволяет создать объект `maths_vector` с помощью конструктора `variant`.

Третья форма предназначена для внедрения имен, определенных в другом месте, в область действия объявления `using`. Например:

```
namespace gdg::maths {
    using std::inner_product;
}
```

После этого внутри пространства имен `gdg::maths` можно вызывать стандартную версию `inner_product` без префикса `std`. Если имя уже существовало во внешней области, оно будет скрыто этим новым объявлением. Если потребуется вызвать другую функцию `inner_product`, ее придется квалифицировать пространством имен.

Объявления `using` более специфичны, чем директивы. Они вводят один символ в текущую область видимости. Однако эти объявления следует использовать с осторожностью и в наиболее ограниченной области видимости. Объявления `using` тоже могут приводить к созданию перегруженных версий, поэтому старайтесь размещать их так, чтобы они были хорошо видны клиентам. Опасность их применения в области видимости файла меньше, чем опасность применения директивы `using`, но риск все же остается.

Четвертая форма предназначена для объявления директив `using`. Она имеет очень ограниченное применение: допустим, вы пишете пример кода для презентации и хотите сообщить аудитории, что используете конкретное пространство имен:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!";
    return 0;
}
```

Чтобы не вводить `std::cout`, вы импортируете все символы из пространства имен `std`, объявленные в заголовке `<iostream>`. Это очень удобно.

Очень, очень удобно.

Это простое применение директивы `using` прекрасно подходит для такого тривиального кода. Но в более сложном коде могут происходить удивительные вещи, потому что директива `using` не только вводит новые символы в текущую область видимости.

Повторим еще раз: директива `using` не только вводит новые символы в текущую область видимости. Она делает нечто гораздо более удивительное, и, чтобы понять происходящее, нужно немного углубиться в теорию графов.

Не загрязняйте
глобальное про-
странство имен: это
ценный ресурс.

КУДА ПОПАДАЮТ СИМВОЛЫ

Во-первых, рассмотрим идею ориентированного ациклического графа (Directed Acyclic Graph, DAG), придуманного коллегой Бьерна Страуструпа — Альфредом Ахо (Alfred Aho). Граф — это набор узлов, связанных отношениями. Ориентированный граф — это набор узлов, связанных односторонними отношениями. Например, иерархия классов — это граф с отношениями «является потомком» и «является родителем» между классами-узлами. Эти отношения имеют противоположные направления. Если уберем отношения «является потомком», то получится ориентированный граф, потому что в нем останутся только односторонние отношения. Ациклический граф имеет вход и выход и не дает возможности вернуться туда, откуда вы начали, следуя по отношениям. Иерархия родительских отношений классов не только ориентирована, но и ациклична.

То же можно сказать о пространствах имен и отношениях включения (вложения). Пространство имен может содержать другое пространство имен, при этом граф вложенности будет ориентированным и ациклическим. Например:

```
namespace World {  
    namespace Buildings {  
        namespace Municipal {}  
        namespace Business {}  
    }  
}
```

Мы можем перейти от `World` к `Municipal`, исследуя вложенные пространства имен, но не можем перейти от `Municipal` к `World` аналогичным путем.

Во-вторых, рассмотрим понятие ближайшего общего предка (Lowest Common Ancestor, LCA). LCA пары узлов — это самый глубокий узел, общий для обоих. В предыдущем примере с пространствами имен оба пространства, `World` и `World::Buildings`, являются общими предками пространств имен `World::Buildings::Municipal` и `World::Buildings::Business`, но ближайшим общим предком является пространство имен `World::Buildings`.

Теперь, познакомившись с понятиями DAG и LCA, посмотрим, что именно делает директива `using`. Соответствующее описание в разделе [namespace.udir]¹ стандарта занимает три страницы. Если говорить кратко, то директива `using` вводит новое смысловое значение в область видимости, которая является LCA текущей области видимости и собственной области видимости целевого пространства имен. Это может показаться удивительным, и этот аспект может вызывать еще большее удивление с ростом базы кода и увеличением количества пространств имен. Давайте рассмотрим реальный пример, используя приведенную выше иерархию пространств имен.

```
namespace Result {
    struct Victory {};
    struct DecisiveVictory : Victory {};
    struct CrushingVictory : DecisiveVictory {};
}

namespace gdg {
    int signal_result(Result::CrushingVictory);
}

namespace World {
    int signal_result(Result::CrushingVictory);
    namespace Buildings {
        namespace Municipal {
            int signal_result(Result::DecisiveVictory);
        }
        namespace Business {
            int compete() {
                using namespace Municipal;
                using namespace gdg;
                return signal_result(Result::CrushingVictory());
            }
        }
    }
}
```

¹ <https://eel.is/c++draft/namespaces.udir>

Мы добавили тип `Victory` в новое пространство имен `Result` и функцию `signal_result`, описывающую результат игры. Давайте посмотрим, что происходит в функции `compete`.

Прежде всего директива `using Municipal` вводит имя `int signal_result(Result::DecisiveVictory)` в LCA пространств имен `Municipal` и `Business`. Напомним, что это действие скрывает предыдущее значение `signal_result`. Сможете ли вы определить, что именно будет скрыто?

Скрыты будут `gdg::signal_result` и `World::signal_result`. Запомните это.

Следующая директива `using` вводит имя `gdg::signal_result` в глобальное пространство имен — LCA пространства имен `gdg` и `Business`. Однако это имя было скрыто именем `World::signal_result`, которое, в свою очередь, было скрыто именем `Municipal::signal_result`. Итак, какая из функций `signal_result` будет вызвана в конце функции `compete`?

`Municipal::signal_result` — единственный доступный выбор. Даже притом, что она принимает параметр типа `DecisiveVictory`, а не `CrushingVictory`, это лучший вариант, потому что он единственный.

Получилось ли у вас следовать за нашими рассуждениями и прийти к правильному ответу?

Я понимаю¹, что этот пример может показаться надуманным и чрезмерно усложненным. И все же он не сильно отличается от примера из реального мира, который я распутывал почти целую неделю. Я убрал весь посторонний код, не имевший ничего общего с ошибкой, чтобы упростить ее поиск. Представьте, что этот код занимает несколько тысяч строк в нескольких файлах. Представьте, что действие этого кода проявляется в виде записи не той победы, которая не слишком далека от правильной победы в игре. Представьте, что эта ошибка связана не с качеством победы, а с особенностями передачи деталей победы остальной части системы. С какой стати вы решили, что `CrushingVictory` не будет записана? Вот она, прямо в коде. Догадка осенила меня, только когда я пролистывал код и вдруг понял, что зашел не туда. Я обвинял отладчик и слишком долго спорил об оптимизаторах со всеми, кто меня слушал. Возможно, секция COMDAT была неправильно свернута. Но в действительности ошибка была обусловлена моим непониманием, как работают директивы `using`.

¹ Своим опытом делится Гай Дэвидсон.

ЕЩЕ БОЛЕЕ КОВАРНАЯ ПРОБЛЕМА

Возможно, никакие наши доводы не убедили вас. Возможно, вы работаете над небольшими проектами и не используете вложенных пространств имен. Однако есть еще одна ловушка, в которую легко попасть. Взгляните на следующий фрагмент кода из разработки 2005 года:

```
// numerical_maths.h
namespace maths {
    float clamp(float v, float max, float min);
}
using namespace maths;

// results.cpp
#include "numerical_maths.h"
#include <algorithm>
using namespace std;
```

`results.cpp` содержит множество вызовов, таких как:

```
int x = clamp(12, 6, 4);
```

Несмотря на отсутствие перегруженной версии, принимающей `int` вместо `float`, существующая функция является подходящим кандидатом, потому что `int` можно преобразовывать в `float`, пусть и с потерей точности при больших значениях. Технически это была ошибка, но она не проявилась ни разу, потому что значения всегда были достаточно малы.

В 2017 году произошло важное событие: в пространство имен `std` была добавлена функция `clamp`. В результате вышло так, что `clamp` в `results.cpp` получила дополнительную перегруженную версию. К сожалению, будучи шаблоном, функция `std::clamp()` оказалась лучшим выбором по сравнению с `maths::clamp()`, когда все параметры имели тип `int`, потому что в этом случае не требовалось выполнять преобразование. К счастью, она делала то же самое, что и `maths::clamp()`. К сожалению, она принимала параметры `max` и `min` в обратном порядке. К счастью, реализация предупреждала, если параметр `max` оказывался меньше параметра `min`. К сожалению, мы отключили все такие предупреждения, потому что из-за них отладка ужасно замедлялась.

В результате начали появляться странные и малозаметные ошибки. Неправильные результаты ограничения диапазона сказывались в самых неожиданных местах. Они не были катастрофическими, но не соответствовали замыслу. Проблема не ограничивалась рамками `results.cpp`: функция `clamp` использовалась очень широко. К счастью, у нас было достаточно времени,

чтобы исправить все ошибки, и в конце все мы вздохнули с огромным облегчением.

На самом деле последнее предложение не соответствует действительности.

Это было не лучшее время, и только когда кто-то, заглянув на сайт cppreference.com, узнал о появлении `std::clamp`, заметил иной порядок параметров и осознал причину непонятных проблем, все фактически стало на свои места. Из-за того что в коде было разбросано несколько тысяч вызовов `clamp`, потребовалось провести утомительный поиск по всему коду, чтобы отловить все эти вызовы и проверить порядок параметров.

Простого перемещения директивы `using` из заголовочного файла во все файлы с исходным кодом было бы недостаточно: возникла бы та же проблема, но наличие пространства имен могло хотя бы сообщить о другом источнике символов. Объявления `using` в начале каждого файла с исходным кодом тоже было бы недостаточно, потому что в некоторых местах инженеры писали код после обновления C++17 и непреднамеренно меняли местами параметры, вызывая `std::clamp`, а не `maths::clamp`. Каждое из этих объявлений было бы признано недействительным.

Единственной защитой от этого ужаса было бы добавление объявления `using` вместо директивы в файлы с исходным кодом, в идеале в самой узкой области видимости, или чтобы полностью квалифицировать использование `clamp`. Помещая директиву `using` в файл с исходным кодом, вы рискуете придать новые значения своим символам, объявленным в других пространствах имен. Если вам повезет, то компилятор сообщит об ошибке `ambiguous symbol` (неоднозначный символ). Если нет, то появится более предпочтительная перегруженная версия с небольшими отличиями от ожидаемого вами выбора.

Мы призываем вас вернуться и перечитать первую часть этой главы.

РЕШЕНИЕ ПРОБЛЕМЫ ОПЕРАТОРОВ РАЗРЕШЕНИЯ ОБЛАСТИ ВИДИМОСТИ

Директивы `using` в глобальной области видимости, бесспорно, удобны, и от этого удобства трудно отказаться. Без них пришлось бы щедро разбрасывать операторы разрешения области видимости `::` по всему коду. Это еще можно терпеть в отношении пространства имен `std`. Но при использовании длинных идентификаторов пространств имен, таких как

`World::Buildings::Business`, код становится сложнее читать и воспринимать. Давайте посмотрим, как правильно использовать пространства имен.

Вспомним о существовании единого глобального пространства имен. В глобальном пространстве имен желательно объявлять только функцию `main` и объявления `extern "C"`. Проще говоря, все свои символы вы должны объявлять в своих пространствах имен.

Пространства имен инкапсулируют символы, а также предлагают возможности абстракции. Абстракции должны быть достаточно короткими, чтобы их проще было воспринимать с одного взгляда. Например, объединение связанных классов в единое пространство имен — это форма абстракции.

Однако по мере роста проектов количество классов будет расти, как и количество пространств имен, если вы разумно относитесь к размерам абстракций. Это приводит к описанным выше объявлениям, таким как `World::Buildings::Business`. Такие идентификаторы требуют вводить много символов с клавиатуры и являются причиной, по которой инженеры предпочитают использовать директивы `using`.

К счастью, существует простое решение: псевдонимы для пространств имен. Например, `World::Buildings::Business` можно сократить до более простого:

```
namespace BizBuild = World::Buildings::Business;
```

Этот псевдоним пространства имен не импортирует никаких символов ни в текущую область видимости, ни в родительскую и не вносит никакой путаницы. Он просто вводит другое имя — псевдоним пространства имен. Добавив приведенное выше объявление, вы сможете вводить такой код:

```
auto x = BizBuild::compete();
```

вместо:

```
auto x = World::Buildings::Business::compete();
```

До появления пространств имен широко применялся шаблон именования, основанный на встраивании названия предметной области в идентификаторы, например, так:

```
int BizBuild_compete();
```

В данном случае подразумевается, что функция `compete` относится к предметной области `Business Buildings`. До сих пор еще можно видеть в программах использование символа подчеркивания для отделения названий

предметных областей от контекстных идентификаторов, хотя в этом нет необходимости, так как есть возможность использовать пространства имен и псевдонимы. Ценой лишнего символа, двойного двоеточия вместо одного подчеркивания вы получаете возможность семантического отделения предметной области от контекстного идентификатора. Это семантическое отделение распознается синтаксическим анализатором C++ и может использоваться инструментами редакторов кода, например, для автоматического завершения символов, перечисления членов пространства имен и т. д.

ИСКУШЕНИЕ И РАСПЛАТА

Порой очень трудно свернуть с пути, начав идти по нему.

Большие проекты в идеале состоят из множества библиотек, каждая из которых объявляет свое пространство имен. Таким способом производится деление области решений на подразделы — полезный прием абстракции. Каждая библиотека будет иметь набор заголовков, определяющих ее возможности через хорошо спроектированный API. Если вы решите следовать за этими объявлениями с помощью директивы `using`, то пользователям вашего кода не придется уточнять какие-либо символы. И тут вы можете решить, что делаете им одолжение, избавляя от ввода лишнего кода.

На первый взгляд эта идея кажется хорошей и здравой, но с увеличением размера базы кода и количества символов возрастает вероятность непреднамеренного создания перегруженных версий. В конце концов здравый смысл возобладает, и вам придется удалить все ваши директивы `using`. Внезапно ваш код перестанет компилироваться, так как компилятору встретятся тысячи неизвестных символов. Единственным решением будет перенести директивы `using` в файлы заголовков клиентов, а потом в файлы реализации, а затем, наконец, заменить их конкретными объявлениями `using`. Эту работу трудно назвать простой или увлекательной.

Возможно, когда-нибудь эта проблема будет решена на уровне инструментов. Мы сможем просматривать исходные файлы, добавлять и удалять полное определение пространств в идентификаторы одним щелчком кнопкой мыши. Будут генерироваться предупреждения при встрече с директивами `using` в самой широкой области видимости и предлагаться подходящие альтернативы для исправления ошибочного исходного кода. Но до тех пор, пожалуйста, следуйте совету в Core Guideline: «SF.7. Не используйте `using namespace` в глобальной области видимости в заголовочном файле».

ПОДВЕДЕМ ИТОГ

- Использование директив `using` в широкой области видимости — рискованный шаг, который может дорого обойтись, приведя к созданию неожиданных перегруженных версий. Лучше используйте объявления `using`.
- Игнорирование этого правила становится особенно опасным с ростом базы кода и может потребовать больших трудозатрат для последующего исправления ситуации.
- Помещайте объявления `using` в самой узкой области видимости, например в определениях классов и функций.

IV

ИСПОЛЬЗУЙТЕ НОВУЮ ОСОБЕННОСТЬ ПРАВИЛЬНО

Глава 4.1 F.21. Для возврата нескольких выходных значений используйте структуры или кортежи.

Глава 4.2 Enum.3. Страйтесь использовать классы-перечисления вместо простых перечислений.

Глава 4.3 ES.5. Минимизируйте области видимости.

Глава 4.4 Con.5. Используйте constexpr для определения значений, которые можно вычислить на этапе компиляции.

Глава 4.5 T.1. Используйте шаблоны для повышения уровня абстрактности кода.

Глава 4.6 T.10. Задавайте концепции для всех аргументов шаблона.

ГЛАВА 4.1

F.21. Для возврата нескольких выходных значений используйте структуры или кортежи

ФОРМА СИГНАТУРЫ ФУНКЦИИ

Знакомясь с функциями, мы узнаем, как принимать аргументы, затем манипулировать входными данными и возвращать выходной результат. Функции способны принимать сколько угодно аргументов, но вернуть могут только одно значение.

Такое существенное ограничение может стать проблемой. Как сообщить вызывающему коду об ошибке? Выше уже говорилось, что эта проблема не имеет удовлетворительного решения. Но один из популярных подходов состоит в том, чтобы принять в аргумент ссылку и записать в указанное по ссылке место код ошибки одновременно с возвращением результата.

Таким образом, мы получаем два типа аргументов функций: входные и выходные параметры. Входные параметры используются для вычислений внутри функции, а выходные – для передачи, кроме возвращаемого значения, дополнительных результатов.

Вследствие этого устоялась типичная практика расположения параметров функции в следующем порядке:

```
возвращаемое_значение идентификатор(входные_параметры, выходные_параметры);
```

Вот пример сигнатуры функции, принимающей некоторые строковые данные и возвращающей их вместе с количеством данных, добавленных в строку:

```
int amend_content(std::string const& format, std::string& destination);
```

Она имеет возвращаемое значение типа `int`, входной параметр `format` и выходной параметр `destination`. Вызывающий код должен создать строку перед вызовом функции и передать ее во втором параметре.

Такой прием создает потенциальную ловушку. Как узнать, где заканчиваются входные и начинаются выходные параметры? Автор должен дополнительно отразить это в документации, чтобы пользователь функции мог правильно упорядочить свои аргументы. Объявление функции должно выглядеть примерно так:

```
int amend_content(std::string const& format,  
                  /*выходной параметр*/ std::string& destination);
```

В этом примере можно подумать, что неконстантная ссылка ясно намекает на возможность записи по ней нового значения. Однако вы не должны делать никаких предположений о возможных действиях и решениях пользователей вашей функции, а эта документация может оказаться лишь дополнительным бременем и рискует быть просто проигнорирована. Добавление дополнительных входных и выходных параметров, например дополнительных деталей формата или канала для возврата ошибки, требует постоянного обновления документации.

Придется также решить, какое из выходных значений должно передаваться клиентам через возвращаемое значение, а какое — через выходной параметр. Кто-то наверняка захочет, чтобы функция возвращала исправленную строку, а не количество добавленных в нее данных. Это может привести к крайне утомительным и бессодержательным спорам между коллегами-разработчиками без реальной пользы, потому что при обсуждении вариантов использования в команде программистов обычно побеждает самый криклиwyй.

С другой стороны, единственное возвращаемое значение прекрасно описывает само себя: оно по определению является выходным значением.

ДОКУМЕНТИРОВАНИЕ И АННОТИРОВАНИЕ

Еще на рубеже веков в Microsoft было предложено решение проблемы документирования природы параметров функций — язык аннотаций исходного кода (Source code Annotation Language, SAL). Аннотации должны определяться в заголовочном файле `sal.h`, распространяться как часть реализации¹ и использоваться в сигнатуре функции для описания входных и выходных параметров. Возьмем для примера хорошо знакомую нам сигнатуру функции `memcpy`:

```
void* memcpy(
    void* dest,
    const void* src,
    size_t count
);
```

Применение к ней аннотаций дает следующее:

```
void* memcpy(
    _Out_writes_bytes_all_(count) void* dest,
    _In_reads_bytes_(count) const void* src,
    size_t count
);
```

Ключевые слова предоставляют дополнительную машиночитаемую информацию о характере параметров. В данном примере `_Out_writes_bytes_all_(count)` сообщает, что в `dest` записывается `count` байт, а `_In_reads_bytes_(count)` — что из `src` извлекается `count` байт. Доступно множество самых разных аннотаций, что делает этот способ очень удобным для устранения неоднозначности в интерпретации параметров.

Упомянутые аннотации реализованы как макросы, которые исчезают на этапе обработки кода препроцессором, но благодаря им инструменты в Microsoft Visual Studio IDE могут гарантировать правильное использование функций. Если вы пользуетесь Visual Studio, то можете попробовать извлечь выгоду из дополнительных средств анализа кода.

Однако SAL не избавляет от необходимости документировать код. Он может помочь автоматизированным средствам поиска ошибок и злоупотреблений, но на самом деле лишь переносит проблему описания параметров в другое место. Эта проблема была унаследована от C, и, несмотря на всю полезность

¹ <https://docs.microsoft.com/ru-ru/cpp/code-quality/using-sal-annotations-to-reduce-cpp-code-defects>

SAL для оформления функций C, в C++ есть другие способы сделать сигнатуры функций более выразительными, включая использование самого языка. Преимущество этих способов в том, что они, выполняя свое прямое назначение, служат цели создания самодокументированного кода.

ТЕПЕРЬ МОЖНО ВЕРНУТЬ ОБЪЕКТ

Одно из лучших решений описанной проблемы — вернуть экземпляр структуры или класса. Вместо того чтобы разбрасывать выходные параметры по сигнатуре вашей функции, можно все значения, вычисленные функцией, упаковать в единый объект и передать вызывающей стороне. Возможность вернуть объект — отличное дополнение к ясности, обеспечиваемой введением конструкторов копирования и перемещения. Однако эта возможность имеет противоречивую историю.

Впервые начав использовать C++, многие программисты были в отчаянии от того, что приходилось возвращать объекты по значению. Ведь было заметно, что для возврата объекта вызывается конструктор копирования, а вслед за ним — оператор присваивания для сохранения результата вызова. В годы, предшествовавшие стандартизации, разные компиляторы по-разному оптимизировали эту последовательность действий, иногда ненадежными способами.

Одной из простых оптимизаций была передача выходного параметра функции, то есть вместо:

```
BigObj create_big_obj(); // Прототип функции
BigObj big_obj = create_big_obj();
```

использовался такой код:

```
void create_big_obj(BigObj&); // Прототип функции
BigObj big_obj;
create_big_obj(big_obj);
```

Вместо возврата и присваивания `BigObj` следовало конструировать и изменять объект. Такой прием давал особенно большие преимущества в производительности, если класс включал контейнер, потому что первый вариант предполагал двойное копирование содержимого этого контейнера.

В 1990-х годах появилась поддержка пропуска копирования. Она объединяла возврат и присваивание в одну операцию. Однако разные компиляторы

действовали по-разному в разных обстоятельствах, поэтому реализация возврата объектов с возможностью последующего переноса на другие платформы была невозможна. Например, одна из проблем с объединением возврата и присваивания состояла в том, что каждая из этих операций могла иметь побочные эффекты, которые теперь, после их объединения, исчезли.

Взгляните на этот класс:

```
class SmallObj {
public:
    SmallObj() {}
    SmallObj(SmallObj const&) {std::cout << "copying\n";}
};
```

Конструктор копирования имеет побочный эффект в виде вывода сообщения в стандартный вывод. Взгляните на этот фрагмент кода:

```
SmallObj so() {
    return SmallObj(); // RVO (см. ниже)
}

int main() {
    std::cout << "Testing...\n";
    SmallObj s = so();
}
```

Выполнение этого фрагмента приведет к одному из следующих результатов, в зависимости от компилятора и даже его настроек:

```
Testing...
copying
copying
Testing...
copying
Testing...
```

Как вы понимаете, это сильно раздражало. Было два типа пропуска копирования: оптимизация возвращаемого значения (Return Value Optimization, RVO) и оптимизация именованного возвращаемого значения (Named Return Value Optimization, NRVO). Первый из них продемонстрирован в примере выше. Если возвращается безымянный временный объект, то компилятор может создать его непосредственно там, куда должно быть скопировано возвращаемое значение.

Второй тип пропуска несколько сложнее в реализации. Если возвращается именованный объект, то при определенных условиях компилятор

может создать его непосредственно там, куда должно быть скопировано возвращаемое значение. Угадывать природу этих состояний было сродни колдовству. Вы могли помочь компилятору, предоставив встроенные (*inline*) определения деструктора и конструктора копирования вместе со всеми членами, но было чрезвычайно легко все испортить и вызвать значительную потерю производительности из-за отказа компилятора выполнить пропуск копирования.

Разочарованные, программисты отказывались от возврата результатов по значению. Со стандартизацией появилось правило «как если бы», которое ясно дало понять, что можно эмулировать требуемое поведение, явно разрешая пропуск копирования, даже при наличии побочных эффектов. Но и в этом случае не было никаких гарантий, снова обнаруживалось, что слишком легко получить провалы в производительности небрежным добавлением переменных-членов.

С выходом следующего стандарта ситуация продолжала улучшаться. C++11 представил семантику перемещения, которая придала пропуску копирования более широкий смысл. Теперь стало возможно определить отдельный конструктор специально для перемещения, а не для копирования объектов. Это сделало возврат объектов из функций гораздо более привлекательным, потому что для возврата начал вызываться конструктор перемещения. Так что даже если вдруг пропуск копирования не случался, то стоимость возврата можно было снизить, особенно для контейнеров.

Для обновления стандарта были предприняты героические усилия, чтобы возврат объектов как можно чаще обходился как можно дешевле. Начиная с C++17 на компиляторы возложена обязанность пропускать вызов конструкторов копирования и перемещения объектов в операторе `return`, даже если это означает исключение наблюдаемых побочных эффектов, когда возвращаемый объект является значением `rvalue`⁴ с типом, соответствующим типу возвращаемого значения функции. И до C++17 это положение поддерживалось большинством компиляторов, но стандартизация этой практики вселила в пользователей уверенность.

Теперь на оптимизацию возвращаемого значения (Return Value Optimization, RVO) можно смело положиться и всегда возвращать объекты из функций путем вызова конструктора, как в приведенном выше примере

⁴ К категории `rvalue` относятся выражения, результатами которых инициализируются объекты и битовые поля, а также выражения, используемые для вычисления значений operandов в операциях в соответствии с контекстом их применения. — *Примеч. пер.*

RVO. Возможно, вы также видели, что `return` можно вызывать со списком инициализации:

```
std::pair<int, int> f(int a, int b) {
    return {a + b, a * b};
}
```

Он также является значением `rvalue`, что позволяет компилятору сгенерировать код в стиле RVO. Необходимость в NRVO отсутствует, так как у объекта нет имени. Такая ситуация не связана с оптимизацией или флагами компилятора: это просто имеет место благодаря фундаментальным изменениям в C++17, касающимся спецификации временных значений и `rvalue`.

Такое положение вещей может сбивать с толку старых программистов, потому что это означает, что компилятор предпочтет пропуск копирования побочным эффектам, и если программист что-то регистрирует в конструкторах перемещения или деструкторах, то его может удивить отсутствие сообщений. И все же это верное решение, соответствующее цели C++: предлагать абстракции с нулевыми накладными расходами. Не следует легкомысленно избегать возможностей повышения производительности.

Для обновления стандарта были предприняты героические усилия, чтобы возврат объектов как можно чаще обходился как можно дешевле.

МОЖНО ТАКЖЕ ВЕРНУТЬ КОРТЕЖ

Учитывая возможность спокойно положиться на возврат данных по значению без потери производительности, можно полностью избавиться от выходных параметров.

Очень распространенный вариант использования выходного параметра и возвращаемого значения — сигнализация об ошибке. Вернемся снова к классу `BigObj`:

```
class BigObj {
public:
    BigObj(int a, int b);

private:
    // реализация
};
```

Возможно, вам приходилось видеть функции с такими прототипами:

```
BigObj do_stuff(int a, int b, error_code& err);
```

Функция принимает некоторые входные данные и ссылку на объект для передачи кода ошибки и возвращает объект. Вызывающий код, прочитав код ошибки, может проверить допустимость созданного объекта и вообще его существование. Возможно, автор функции не хотел или не мог использовать исключения, столкнувшись с ошибкой во время создания BigObj.

Это несколько неэлегантный способ сообщения об ошибках, потому что вызывающей стороне предлагается два канала передачи данных: через возвращаемый объект и через ссылку `error_code`. Существуют и другие рекомендации по обработке ошибок, обсуждавшиеся в главе 3.5. Они выступают против такого подхода. Одна из рекомендаций предлагает возвращать вариант, способный нести BigObj и `error_code`. Однако иногда бывает желательно сигнализировать о достоверности возвращаемого объекта, передавая специальный нулевой код ошибки, обозначающий отсутствие ошибки.

Учитывая, что вы теперь можете безопасно возвращать объекты по значению, также смело можете вернуть и пару `std::pair`. Вот пример такой функции:

```
std::pair<BigObj, error_code> do_stuff(int a, int b) {
    // проверка допустимости входных значений
    return { {a, b}, {error} };
}
```

Оператор `return` создает значение `rvalue`, поэтому копирование и присваивание могут быть пропущены механизмом RVO. Стандартная библиотека использует этот прием в нескольких местах. Функция-член `insert` в ассоциативных контейнерах возвращает значение типа `std::pair<iterator, bool>`, сообщающее о том, была ли выполнена вставка и в какое место.

На самом деле вы не ограничены типом `std::pair`, точно так же можно вернуть `std::tuple` любого размера. Существует удобный библиотечный механизм для связывания элементов `std::tuple` с отдельными объектами под названием `std::tie`, появившийся в C++11. Он позволяет объявлять объекты, а затем привязывать их к `std::pair` или `std::tuple`, например:

```
BigObj bo;
error_code ec;
std::tie(bo, ec) = do_stuff(1, 2);
```

Мы знаем, о чём вы подумали: это просто объявление объекта с последующим присваиванием ему значения, и означает оно возврат к старым недобрым дням конструирования и присваивания вместо пропуска копирования.

И вы абсолютно правы, именно поэтому в C++17 были введены структурные привязки. Благодаря поддержке этой особенности в языке и в стандартной библиотеке вы можете поместить возвращаемые значения непосредственно в новые объекты, например:

```
auto [bo, ec] = do_stuff(1, 2);
```

Здесь можно видеть, как `tie` был заменен синтаксическим сахаром в виде `auto` и квадратных скобок.

Структурное связывание — любимая многими возможность в C++17. Её можно использовать даже для привязки к массиву или структуре:

```
int a[2] = {1, 2};
auto [b, c] = a; // b и c — целые числа int, где b = 1 и c = 2
struct S {
    int a;
    int b;
};
S make_s();

auto [d, e] = make_s(); // d = первому члену возвращаемой структуры
// e = второму члену
```

Вариантов использования выходных параметров немногого. Конечно, возврат кортежей — это ловкий трюк, но не забывайте возвращать структуру. Если возвращаемые значения каким-то образом связаны, то, возможно, вы находитесь в процессе поиска абстракции. Для возврата структуры используются те же приемы, что и для возврата кортежа. Пропуск копирования означает, что значения создаются там, где вызывающая сторона ожидает их найти.

Всегда проверяйте, не обходите ли вы стороной абстракцию, когда просто возвращаете кортеж. Пара, состоящая из `BigObj` и кода ошибки, не поддается разумной компоновке в новую абстракцию, но кортеж, состоящий из нескольких связанных объектов, скорее всего, нуждается в выборе имен для отдельных частей. Представьте строку и количество символов в ней: здесь явно проглядывает небольшая абстракция с именем, например, `modified_string`, ожидающая своего воплощения.

ПЕРЕДАЧА И ВОЗВРАТ ПО НЕКОНСТАНТНОЙ ССЫЛКЕ

В начале этой главы мы говорили о входных и выходных параметрах. Однако существует третий тип параметров, которые называют входными/выходными. Подобно выходным параметрам, они также являются ссылками или указателями на неконстантные объекты. Обычно входные/выходные параметры располагаются в объявлении функции между входными и выходными параметрами, соответственно можно расширить прототип функции, описанный в начале, как показано ниже:

```
возвращаемое_значение идентификатор(входные_параметры,
входные/выходные_параметры, выходные_параметры);
```

Суть этой рекомендации состоит в том, чтобы отделить входные данные от выходных, сделать так, чтобы входные данные находились между скобками в объявлении функции, а выходные ограничивались типом возвращаемого значения. Можно ли применить рассматриваемую в этой главе рекомендацию к входным/выходным параметрам?

Рассмотрим объект `Report`, собирающий данные из различных источников. Каждый источник предоставлен функцией, принимающей объект `Report` и некоторые дополнительные параметры и возвращающей количество элементов, добавленных в объект. Например:

```
class Report { ... };

int report_rendering(int, int, Report&); // входные параметры
int report_fileIO(int, int, int, Report&); // затем входные/выходные
                                             // параметры

std::pair<Report, int> collect_report(const char* title) {
    auto report = Report(title);
    int item_count = report_rendering(1, 2, report);
    item_count += report_fileIO(0, 0, 1024, report);
    return {report, item_count};
}
```

Всегда проверяйте,
не обходите ли вы
стороной абстрак-
цию, когда просто
возвращаете кортеж.

Чтобы реализовать возврат выходных значений в виде единого объекта, сигнатуры функций должны выглядеть так:

```
std::pair <Report, int> report_rendering(int, int, Report&);
std::pair <Report, int> report_fileIO(int, int, int, Report&);
```

Это оказалось бы неприятный эффект на код:

```
std::pair<Report, int> collect_report(const char* title) {
    auto report = Report(title);
    int item_count = report_rendering(1, 2, report).second;
    item_count += report_fileIO(0, 0, 1024, report).second;
    return {report, item_count};
}
```

Обращение к члену `.second` выглядит уродливо, при этом первый элемент в паре отбрасывается. Вызывающая сторона уже имеет объект, так как именно она передала его в функцию, так зачем его возвращать?

На самом деле если восстановить возвращаемые типы в прототипах функций и переместить входной/выходной параметр в позицию перед входными параметрами, то мы получим:

```
int report_rendering(Report&, int, int);
int report_fileIO(Report&, int, int, int);
```

Эти функции получают объект и изменяют его в соответствии с входными параметрами. Они выглядят как функции-члены, особенно если сравнить их сигнатуры с сигнатурами потенциальных функций-членов:

```
int Report::report_rendering(int, int);
int Report::report_fileIO(int, int, int);
```

Реализация передачи входных и выходных параметров в дополнение к входным параметрам на самом деле всего лишь способ расширения API класса без создания новой функции-члена. Это прием, который оценят по достоинству клиенты, использующие класс: ведь он не требует обращения к автору кода с просьбой добавить новые возможности. Мы тоже, со своей стороны, хотим отметить важность этого приема: он помогает обеспечивать поддержку основного правила «С.4. Преобразуйте функцию в функцию-член, только если ей нужен прямой доступ к представлению класса».

Рассматриваемая рекомендация способствует созданию минималистичных и одновременно полных интерфейсов. Если функцию можно определить как функцию, не являющуюся членом, то в интерфейсе вашего класса может стать на одну функцию меньше. Как вариант, если у вас есть коллекция дружественных функций, не являющихся членами, или коллекция геттеров и сеттеров, которые вызываются несколькими функциями, а они могут оказаться дружественными функциями, то это означает, что вы определили

кандидатов для расширения вашего интерфейса. Если вы можете исправить этот дисбаланс, исправьте его.

С другой стороны, как оказывается, существует вариант получения и возврата объектов по неконстантной ссылке: библиотека `iostream`. Оператор шеврона (`<<`) для вывода символа выглядит так:

```
template<class Traits>
std::basic_ostream<char, Traits>& operator<<(
    std::basic_ostream<char, Traits>& os, const char* s);
```

Здесь `basic_ostream` передается и возвращается, что облегчает объединение последовательных вызовов в один оператор, например:

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

Его можно было бы переписать так:

```
operator <<(std::cout, "Hello, world!").operator <<(std::endl);
```

Только первый вызов не является функцией-членом. Второй вызов — это функция-член, потому что `endl` — это указатель на функцию, а в `basic_ostream` есть оператор-член `<<`, принимающий указатели на функции этого типа. Первая форма записи явно удобнее и проще. Чтобы разобраться со смешиванием типов в цепочке во второй форме записи, требуется время, а это создает ненужную когнитивную нагрузку.

Вы имеете полное право разрабатывать API, поддерживающие такой цепочечный синтаксис. Если вам требуется выполнить некоторую последовательность операций, не генерирующих исключений, то реализация чего-то подобного показанному ниже может показаться детской забавой.

```
class Obj { ... };
Obj object;

object.f1(a, b, c) // Все функции-члены возвращают Obj&
    .f2(d, e, f)
    .f3(g, h);
```

Наконец, хотелось бы отметить, что, обнаружив в своем коде передачу функции нескольких входных/выходных параметров, вы можете быть уверены, что упустили из виду некоторую абстракцию. Возможно, вы пытаетесь выполнить своеобразную двойную (или более) диспетчеризацию и смоделировать взаимодействия нескольких классов, что в принципе не совсем правильно.

ПОДВЕДЕМ ИТОГ

Старый совет упорядочивать параметры в порядке «входные, входные/выходные и выходные» потерял свою актуальность благодаря достижениям C++.

- Обязательный пропуск копирования при возврате значений `prvalue` означает, что тщательная проработка интерфейса конструктора может способствовать оптимальному возврату по значению.
- Если нужно вернуть несколько значений, возвращайте их в виде структуры или кортежа.
- Структурное связывание предпочтительнее `tie`, потому что является средством языка, а не библиотеки. Кроме того, этот прием избавляет от необходимости конструировать объект по умолчанию, который будет связан с возвращаемым значением.
- Входные/выходные параметры могут расширять API класса и должны располагаться перед входными параметрами.
- Возвращаемые входные/выходные параметры можно использовать для последовательного вызова функций в стиле оператора шеврона, правда, чтобы сделать код разборчивым, придется приложить усилия.
- Наличие нескольких входных/выходных параметров должно вызывать тревогу и может намекать на незамеченную абстракцию.

ГЛАВА 4.2

Enum.3. Страйтесь использовать классы-перечисления вместо простых перечислений

КОНСТАНТЫ

Константы прекрасны. Типы прекрасны. Константы определенного типа действительно прекрасны. Вот почему классы-перечисления — просто чудо.

Исторически константы определялись как макросы препроцессора, и мы надеемся, что вы больше не чувствуете в них необходимости. В исходном коде, реализующем геометрические вычисления, вы могли видеть нечто подобное:

```
#define PI 3.1415926535897932385
```

А если вам не повезло, то в другом файле вы могли увидеть нечто такое:

```
#define PI 3.1415926 // достаточно для float, недостаточно для double
```

Или такое:

```
#define PI 3.1415926535987932385 // с ошибкой
```

Или такое:

```
#define Pi 3.1415926535897932385 // немножко другое имя
```

Макросы могли даже определиться в заголовочных файлах и напрочь испортить вам день. Эти символы препроцессора не имеют ни типа, ни области

действия. Они просто лексически заменяются на этапе обработки препроцессором. Одной из первых побед C++ стало осознание возможности объявлять объекты с типами, квалифицированными как `const` (пожалуйста, никогда не называйте их константными переменными) и с ограниченной областью видимости. Единое удачное определение числа `pi` было долгожданным событием. На самом деле, начиная с C++20, у нас есть стандартное определение числа `pi`. Оно находится в заголовке `<numbers>` и определено в пространстве имен `std::numbers`:

```
template <>
inline constexpr double pi_v<double> = 3.141592653589793;
inline constexpr double pi = pi_v<double>;
```

Некоторые константы играют важную роль, но имеют произвольные значения. В отличие от PI, E или `MONTHS_IN_YEAR` для представления идей иногда бывает нужно определить несколько именованных значений зачастую в форме небольших целых чисел, таких как 1 для редактирования, 2 для просмотра, -1 для выхода и т. д. Все еще можно встретить код с огромным количеством макросов, определяющих взаимосвязанные целые числа. Вот выдержка из заголовка `WinUser.h`, входящего в состав Windows SDK:

| | |
|---|---------------------|
| <code>#define WM_CTLCOLORSCROLLBAR</code> | <code>0x0137</code> |
| <code>#define WM_CTLCOLORSTATIC</code> | <code>0x0138</code> |
| <code>#define MN_GETHMENU</code> | <code>0x01E1</code> |
| | |
| <code>#define WM_MOUSEFIRST</code> | <code>0x0200</code> |
| <code>#define WM_MOUSEMOVE</code> | <code>0x0200</code> |
| <code>#define WM_LBUTTONDOWN</code> | <code>0x0201</code> |

Почему существует разрыв между `0x01E1` и `0x0200`? Скорее всего, это обусловлено сменой предметной области после `MN_GETHMENU` и отсутствием гарантий, что в дальнейшем не потребуется добавить новые значения. По второму нюансу¹ легко было определить предметную область. А может быть, это получилось совершенно произвольно. Мы никогда этого не узнаем. Невозможно получить эту информацию с помощью простых определений препроцессора.

Перечисляемые типы дают возможность объединить константы, что делает их идеальными для идентификации, например, значений ошибок, и тем самым создать определенную абстракцию. Вместо объявления:

¹ Ниболл (nibble) — полубайт, тетрада или гексадецит (hexadecit – hexadecimal digit), единица измерения информации, равная четырем двоичным разрядам (битам), удобна тем, что представима одной шестнадцатеричной цифрой. — Примеч. пер.

```
#define OK = 0
#define RECORD_NOT_FOUND = 1
#define TABLE_NOT_FOUND = 2
```

можно определить перечисление:

```
enum DB_error {
    OK,
    RECORD_NOT_FOUND,
    TABLE_NOT_FOUND
};
```

Члены этого перечисления получают те же значения, что и константы препроцессора, потому что перечисления по умолчанию начинаются с нуля и увеличиваются на единицу с каждым новым членом. Имена членов перечисления в этом примере записаны прописными буквами, потому что они служат прямой заменой константам препроцессора. Однако это исключение из правил, и имена членов перечислений обычно должны записываться строчными буквами. Впрочем, это вопрос стиля, а не часть стандарта: следование этому стилю помогает не путать символы препроцессора, которые обычно записываются прописными буквами.

К сожалению, ключевое слово `enum` не определяет ни область видимости, ни базовый тип. Это может привести к некоторым интересным проблемам. Рассмотрим перечисление двухбуквенных кодов штатов США:

```
enum US_state {
    ...
    MP, // Отличный вопрос для викторины
    OH,
    OK, // Ого!
    OR,
    PA,
    ...
};
```

Поскольку фигурные скобки не определяют область видимости, как можно было бы ожидать, член `OK` теперь является неоднозначным идентификатором. Если подобные перечисления определены в не связанных друг с другом областях видимости, то это не проблема. В противном случае следует изменить имя члена перечисления, чтобы устраниТЬ неоднозначность. `OK`, очевидно, чрезвычайно полезный идентификатор, поэтому нельзя допустить, чтобы он существовал в таком виде. В мире до C++11 вы наверняка сталкивались с такими членами перечислений, как `S_OK`, `R_OK`, `E_OK` и т. д., делающими код неразборчивым. Однако однобуквенные префиксы были

роскошью, доступной только самым крупным игрокам. В примере выше вы, скорее всего, использовали бы `DBE_OK` или `USS_OK`. Но, украсив префиксом один член перечисления, вы почувствуете себя обязанным украсить и все остальные, в результате чего по всему вашему коду будут разбросаны трехбуквенные аббревиатуры с символами подчеркивания, предшествующими всем членам перечисления.

К счастью, несмотря на неудобства, это препятствие проявится во время компиляции в виде простой ошибки, которую легко устраниТЬ, просто еще немного изуродовав конфликтующие члены перечислений. Другая, более коварная, проблема заключается в неявном преобразовании. Функции могли бы с радостью возвращать член `OK` или, что более вероятно, `DBE_OK`, и вызывающий код мог бы преобразовать его в целое число. То же верно и наоборот: вы могли бы передать член перечисления функции, принимающей целое число. Это приводит к интересным ошибкам, когда вы передаете член одного перечисления, а функция интерпретирует его как член другого перечисления.

ПЕРЕЧИСЛЕНИЯ С ЗАДАННОЙ ОБЛАСТЬЮ ВИДИМОСТИ

Стандарт C++11 расширил ключевое слово `enum` и добавил в него две новые возможности. Первая — ограничение области видимости перечисления. Теперь в объявление перечисления можно добавить ключевое слово `struct` или `class`:

```
enum class DB_error { // Ограниченнaя область видимости,
    // и идентификаторы в нижнем регистре...
    OK, // ...за исключением OK, этот идентификатор набран прописными буквами
    record_not_found,
    table_not_found
};

enum struct US_state {
    ...
    MP, // Северные Марианские острова, раз уж вы спросили...
    OH,
    OK,
    OR,
    PA,
    ...
};
}
```

Перечисления с ограниченной областью видимости требуют использовать префикс перечисления с именами членов и тем самым устраняют неоднозначность из-за совпадений имен членов в разных перечислениях. Используя член перечисления с ограниченной областью видимости, вы должны явно указать область с помощью оператора разрешения, например, так:

```
static_assert(DB_error::OK != US_state::OK);
```

Только этот пример не будет компилироваться, потому что операнды слева и справа от оператора `!=` имеют разные типы и необходимо реализовать перегруженную версию `operator !=`.

В стандарте не оговаривается, когда использовать `struct` или `class`. Можно, скажем, использовать `class`, когда определяете другие операции для перечисления. Например, рассмотрим перечисление с днями недели:

```
enum class Day {  
    monday,  
    tuesday,  
    wednesday,  
    thursday,  
    friday,  
    saturday,  
    sunday  
};
```

Вероятно, вам понадобится возможность переходить от одного дня недели к другому, и вы решите определить префиксный оператор инкремента:

```
constexpr Day operator++(Day& d) {  
    switch (d) {  
        case Day::monday:    d = Day::tuesday;    break;  
        case Day::tuesday:   d = Day::wednesday;  break;  
        case Day::wednesday: d = Day::thursday;   break;  
        case Day::thursday:  d = Day::friday;     break;  
        case Day::friday:    d = Day::saturday;   break;  
        case Day::saturday:  d = Day::sunday;     break;  
        case Day::sunday:    d = Day::monday;    break;  
    }  
    return d;  
}
```

```
Day today = Day::saturday;  
Day tomorrow = ++today;
```

БАЗОВЫЙ ТИП

Перечисления имеют базовый тип, который можно задать в точке объявления или определения. Одно из преимуществ такого подхода заключается в возможности, начиная с C++11, объявлять перечисления без определения, поскольку размер типа можно вывести. Если базовый тип не указан, используется значение по умолчанию. Такое перечисление нельзя объявить предварительно. Базовый тип по умолчанию зависит от вида перечисления — с ограниченной или неограниченной областью видимости.

Если перечисление не имеет области видимости, то базовым типом будет целочисленный тип, определяемый реализацией. Он способен представить все значения членов перечисления. В нашем примере с перечислением дней недели значения членов изменяются от нуля до шести, поэтому можно ожидать, что базовым типом будет `char`. Если перечисление имеет ограниченную область видимости, то базовым типом будет `int`. Это может показаться немного расточительным. В типичной реализации вполне достаточно типа `char`.

Для указания базового типа в C++11 был добавлен новый синтаксис:

```
enum class Day : char {
    monday,
    tuesday,
    wednesday,
    thursday,
    friday,
    saturday,
    sunday
};
```

Этот синтаксис можно использовать для определения обоих видов перечислений — с ограниченной и с неограниченной областью видимости. Однако указывать базовый тип следует только в ситуациях, когда это действительно необходимо; экономия трех байт будет заметна, только если в программе имеются тысячи экземпляров объектов, хранящих экземпляры `Day`. Определения с типом по умолчанию проще читать и записывать. С другой стороны, указание типа может способствовать совместимости на уровне ABI.

Перечисления часто используются для определения констант со значениями, равными степеням двойки или определяющими битовые маски. Например:

```
enum personal_quality {
    reliable = 0x00000001,
```

```
warm      = 0x00000002,  
punctual  = 0x00000004,  
...  
generous  = 0x40000000,  
thoughtful = 0x80000000  
};
```

Базовым типом мог бы быть `int`, если бы перечисление заканчивалось членом `generous`, но для представления члена `thoughtful` требуется целое без знака. В общем случае не следует указывать значения членов перечислений: это может привести к опечаткам и ухудшить производительность операторов `switch`. Однако данный пример — исключение.

До публикации стандарта C++11 в этой части существовало что-то вроде серой зоны. Некоторые реализации разрешали предварительное объявление перечислений, ограничивая размер их членов 32 битами, если только они не превышали максимального представляемого значения. Однако переносимость не гарантировалась. Такие серые зоны, отданные на откуп реализациям, были большой проблемой: вы должны были сами определить, как все ваши целевые реализации определяют эти типы. Впрочем, эта проблема не исчезла и сейчас: базовый тип для перечислений с неограниченной областью видимости, в которых тип не указан явно, до сих пор определяется реализацией.

НЕЯВНОЕ ПРЕОБРАЗОВАНИЕ

Еще одна особенность перечислений без области видимости — возможность свободного преобразования их членов в тип `int`. Распространенной ошибкой в прошлом была передача члена перечисления в функцию, принимающую целое число. Это считалось приемлемой и общепринятой практикой, но ведь надо учитывать, что член перечисления может иметь разные значения в разных областях видимости.

```
// territory.h  
enum US_state { // Перечисление без области видимости  
    ...  
    MP, // Есть еще четыре территории  
    OH,  
    OK,  
    OR,  
    PA,  
    ...
```

```

};

...
void submit_state_information(int); // Хм-м, штат США или государство?

// US_reporting.cpp
submit_state_information(OH); // OH – член перечисления
                             // без области видимости

```

При общем удачном стечении обстоятельств `submit_state_information` действительно принимает штаты США, а не государства. К сожалению, с этим API нельзя быть уверенными.

Вы по-прежнему можете преобразовывать перечисления с ограниченной областью действия в базовый тип, но это нужно делать явно с помощью `static_cast` или `std::underlying_type`:

```

// territory.h
enum struct US_state { // Перечисление с областью видимости
    ...
    MP,
    OH,
    OK,
    OR,
    PA,
    ...
};

...
void submit_state_information(int);

// US_reporting.cpp
submit_state_information(static_cast<int>(US_state::OH));

```

Явное приведение демонстрирует, что вы берете на себя ответственность за последствия этого потенциально опасного действия. В вызове выше мы решили, что `submit_state_information` принимает штат США. Приведение — всегда подходящее место для начала поиска странных ошибок.

Также член перечисления можно привести к типу `int`:

```

// US_reporting.cpp
submit_state_information(int(US_state::OH));

```

Конечно, эта проблема обусловлена плохой продуманностью API. Функция должна быть более явной в отношении своего параметра:

```

void submit_state_information(US_state);

```

К сожалению, иногда бывает неразумно сидеть и молча догадываться, о чем думали ваши коллеги при разработке кода, который вы используете, и самый надежный подход — обратиться за разъяснениями.

ПОДВЕДЕМ ИТОГ

Вместо простых перечислений (с неограниченной областью видимости) лучше использовать перечисления-классы (с ограниченной областью видимости), чтобы извлечь выгоду из определенности базового типа по умолчанию и устраниТЬ неоднозначности между символами.

Простое добавление ключевого слова `class` ко всем объявлениям `enum` обходится недорого, а повышает безопасность, запрещая неявное преобразование. Оно в то же время улучшает читаемость, позволяя убрать «украшения» (префиксы) из идентификаторов членов перечислений, ведь те были добавлены только для того, чтобы члены перечислений было проще распознать.

ГЛАВА 4.3

ES.5. Минимизируйте области видимости

ПРИРОДА ОБЛАСТИ ВИДИМОСТИ

«Область видимости» — еще один из перегруженных терминов. Он пришел из информатики, но каждый язык программирования вносит в него свои маленькие особенности. В частности, в языке C++ область видимости — это свойство объявлений в коде и место, где пересекаются видимость и время жизни.

Все объявления в программе находятся в одной или нескольких областях видимости. Области могут вкладываться друг в друга, как и пространства имен, и в большинстве случаев новая область видимости начинается с определения идентификатора. Имена видны только в области, в которой они объявлены, но время жизни объектов не всегда ограничивается областью видимости их имен. Это несовпадение часто сбивает инженеров с толку, когда речь заходит о динамически создаваемых объектах.

Детерминированное уничтожение объектов — выдающаяся особенность C++. Когда имя объекта класса с автоматическим классом хранения выходит за рамки области видимости, то объект уничтожается. Вызывается его деструктор, и все занятые им ресурсы освобождаются. Нет необходимости ждать, пока сборщик мусора сотворит свое волшебство и подметет пол, как в управляемых языках. Тем более что его работа может приводить к неприятным и непредсказуемым побочным эффектам, таким как нехватка ресурсов в неподходящий момент или полное невыполнение операций по очистке. Однако, когда имя простого указателя выходит за пределы области видимости, уничтожается именно указатель, а не объект, на который он указывает. В результате объект может остаться в памяти без имени и, следовательно, будет потеряна всякая возможность вызвать его деструктор.

Время жизни динамически созданного объекта слабо связано с областью видимости его имени. Указатель можно связать с другим именем и обеспечить его сохранность после выхода за границы области видимости имени, к которому он был первоначально привязан. Его можно связать со множеством имен и заработать головную боль, когда потребуется решить вопрос об уничтожении объекта. Вот почему у нас есть класс `std::shared_ptr`. Когда последнее имя, связанное с объектом `std::shared_ptr`, выходит за границу области видимости, то объект, на который ссылается этот указатель, уничтожается. Мы видели это при обсуждении рекомендации «I.11. Никогда не передавайте владение через простой указатель (`T*`) или ссылку (`T&`)». Для подсчетов этот тип использует подсчет ссылок. Счетчик ссылок увеличивается, когда объект `std::shared_ptr` связывается с новым именем, и уменьшается, когда имя покидает область видимости.

Важно помнить, что область видимости — это свойство имен, а не объектов. Объекты имеют время жизни, точнее, класс хранения. Существует четыре класса хранения: статический, динамический, автоматический и локальный для потока. Сколько областей видимости вы можете назвать? Больше двух? На самом деле их шесть:

- блок;
- пространство имен;
- класс;
- параметр функции;
- перечисление;
- параметр шаблона.

Рассмотрим их по порядку.

ОБЛАСТЬ ВИДИМОСТИ БЛОКА

Область видимости блока, вероятно, первая, о которой вы подумали, хотя, возможно, не вспомнили ее названия. Блок или составной оператор — это последовательность операторов, заключенных в фигурные скобки, как в следующем примере:

```
if (x > 15) {  
    auto y = 200;
```

```

auto z = do_work(y);
auto gamma = do_more_work(z);
inform_user(z, gamma);
}

```

Тело оператора `if` — отличный пример области видимости блока. Этот пример имеет небольшой масштаб, и в нем хорошо видно происходящее: сначала объявляются переменные `y`, `z` и `gamma`, а затем они уничтожаются в конце области видимости после возврата из вызова `inform_user`. Области видимости блока можно увидеть в определениях функций, в операторах управления и т. д.

Области могут быть вложенными. Чтобы вложить одну область видимости блока в другую, достаточно добавить еще одну пару фигурных скобок:

```

if (x > 15) {
    auto y = 200;
    auto z = do_work(y);
    {
        // Новая область видимости
        auto y = z;           // Начало действия нового имени y
        y += mystery_factor(8); // Ссылка на это новое имя y
        inform_user(y, 12);   // Еще одна ссылка на новое имя y
    }                      // Конец области видимости нового имени y
    y = do_more_work(z);   // Продолжение области видимости
                           // первого имени y
    inform_user(z, y);
}

```

Этот пример демонстрирует фрагментированную область видимости и некоторые интересные особенности. Символ `y` был «заслонен» объявлением объекта с тем же именем во вложенной области видимости. Первый идентификатор `y` действует во внешней области видимости и становится недоступен, как только объявляется новый идентификатор во вложенной области видимости. Он становится доступным опять с окончанием вложенной области видимости. Все это означает, что его область видимости фрагментирована, то есть не является непрерывной.

Это менее «здравый» пример, чем предыдущий. Такой код, безусловно, допустим, хотя в общем случае неразумно повторно использовать одни и те же имена во вложенных и внешних областях видимости. Такое часто случается при вставке фрагментов кода из других мест и становится одной из причин, почему вставка кода является крайней мерой. Внешняя и вложенная области видимости занимают менее дюжины строк, они, в принципе, легко читаются, но рано или поздно понадобится изменить

или дополнить код. Когда до этого дойдет, есть вероятность, что автор перепутает два объекта с именем `y`.

Некоторые реализации предупреждают о подобных случаях маскировки имен. Это, в свою очередь, довольно сильно сказывается на команде разработчиков, если в ней принята политика устранения всех предупреждений. Упорный отказ отключить предупреждения приводит к тому, что после перехода на версию компилятора с предупреждениями придется исправлять все такие случаи, а уж в процессе этой работы, бывает, обнаруживается на редкость большое количество ошибок.

ОБЛАСТЬ ВИДИМОСТИ ПРОСТРАНСТВА ИМЕН

Как и область видимости блока, область видимости пространства имен начинается после открывающей фигурной скобки, следующей за идентификатором пространства имен:

```
namespace CG30 { // Область видимости начинается здесь
```

Любой символ, объявленный до соответствующей закрывающей скобки, виден, начиная с точки его объявления. В отличие от области видимости блока, такой символ остается видимым после закрывающей скобки во всех последующих определениях того же пространства имен, например:

```
namespace CG30 {  
    auto y = 76; // Начало области видимости у  
} // Здесь область видимости у прерывается  
...  
namespace CG30 { // Продолжение области видимости у  
    auto x = y;  
}
```

Даже притом, что область видимости `у` прерывается, вы все еще можете ссыльаться на этот символ, квалифицировав его с помощью оператора разрешения области `::` и идентификатора пространства имен. Чтобы обратиться к `у`, объявленному в пространстве имен `CG30`, ссылку следует записать как `CG30::у`.

Есть один случай, когда область видимости пространства имен не начинается после открывающей фигурной скобки, и это касается самой внешней области. С началом этой единицы трансляции начинается пространство имен, называемое глобальным. Обычно эту область называют областью

видимости файла или глобальной областью видимости, но это пережитки языка С. Теперь, введя понятие пространства имен, мы можем использовать более точное название.

Поскольку глобальное пространство имен никогда не прерывается, объявленные в нем символы видны повсюду. Объявлять в глобальном пространстве имен что-либо чрезвычайно удобно. Кроме, конечно, других пространств имен, за исключением `main()` и перегрузок операторов, типы операндов которых объявлены в разных пространствах имен. Итак, это чрезвычайно удобно и так же чрезвычайно ужасно. Глобальные объекты — это плохо. Понятно?

Существует еще одно специальное пространство имен, называемое анонимным. Символы, объявленные в анонимном пространстве имен, остаются доступными до конца вмещающей области видимости и имеют внутреннюю привязку. Например:

```
namespace {
    auto a = 17; // приватная переменная для текущей единицы трансляции
}
```

Коль скоро речь зашла о привязке, необходимо уточнить различия между областью видимости, классом хранения и привязкой. Важно четко понимать разницу между именем и объектом. Имя имеет область видимости, которая определяет, когда это имя доступно без уточнения области. Объект имеет класс хранения, который определяет время жизни объекта. Привязка связывает объект с именем.

Объекты со статическими или локальными для потока классами хранения также имеют внутреннюю или внешнюю привязку. Внутренняя привязка делает объект недоступным из другой единицы трансляции. Объекты с автоматическим классом хранения не имеют привязки.

Объекты с динамическим классом хранения не привязаны к имени и не имеют привязки. Они привязываются к указателю, который сам привязывается к имени. Эта косвенность часто вызывает проблемы, выражаяющиеся в виде утечек памяти, но она же обеспечивает превосходную производительность С++: инженер может точно запланировать время жизни объекта, а не полагаться на сборку мусора.

Если открыть анонимное пространство имен в глобальной области видимости, то все символы будут доступны до конца единицы трансляции.

Это может доставлять неприятности, если анонимное пространство имен открывается в заголовочном файле: единица трансляции обычно заканчивается намного позже окончания директивы `#include`. Кроме того, если заголовочный файл используется повторно, то вы получите несколько экземпляров одной и той же сущности, что может не совпадать с вашей задумкой. Если вы собираетесь открыть анонимное пространство имен в глобальной области видимости, то не делайте этого в заголовочном файле.

```
namespace CG30 {
    auto y = 76;
    namespace {
        auto x = 67;
    } // x остается в области видимости
    auto z = x;
} // здесь область видимости x прерывается

namespace {
    constexpr auto pi = 3.14159f;
} // pi остается в области видимости
```

Последняя область видимости пространств имен, которую следует рассмотреть, — это область видимости встроенного пространства имен. Подобно анонимному пространству имен, область видимости символов, объявленных во встроенном пространстве имен, прерывается не в конце этого пространства имен, а в конце вмещающего пространства имен, например:

```
namespace CG30 {
    auto y = 76;
    inline namespace version_1 {
        auto x = 67;
    } // имя x все еще видимо
    auto z = x;
} // Область видимости x прерывается здесь
```

Как видите, пространства имен — это немного больше, чем кажется на первый взгляд. Область видимости глобального пространства имен — это максимально возможная область, поэтому под минимизацией областей видимости подразумевается отказ от объявления любых символов в глобальной области. Здесь можно заметить удачное совпадение рекомендаций. Кроме того, минимизация областей видимости пространств имен упрощает восприятие содержимого. Длинные и беспорядочные пространства имен теряют свою связность, их следует избегать.

ОБЛАСТЬ ВИДИМОСТИ КЛАССА

Область видимости класса — еще одна разновидность области видимости блока. Область видимости символа, объявленного в определении класса, начинается в точке его объявления и простирается до конца определения класса. Она включает все аргументы по умолчанию для параметров функций-членов, спецификации исключений и тела функций-членов:

```
class RPM {
    static constexpr int LP = 33;
    static constexpr int Single = 45;

public:
    static constexpr int SeventyEight = 78;
    int RotorMax(int x, int y = LP);
}; // RotorMax, LP, Single и SeventyEight остаются доступными
// в функциях-членах класса RPM

int RPM::RotorMax(int x, int y)
{
    return LP + x + y;
}
```

Область видимости `SeventyEight` прерывается, но вы все равно сможете ссылаться на эту константу, явно разрешая область видимости, потому что она является общедоступным членом класса. Сделать это можно с помощью все того же оператора разрешения области видимости `::`, записав ссылку как `RPM::SeventyEight`. Оператор разрешения области видимости служит круг поиска имени для компилятора.

Что понимается под минимизацией области видимости класса? Это означает, что интерфейс должен быть минимальным и полным. С разрастанием интерфейс теряет согласованность.

Мы все видели эпический интерфейс: единственный класс в проекте, служащий домом для беспризорников с такими именами, как `Manager`, `Globals` или `Broker`, и псевдонимами, такими как `TheBlob`. Классы с недостаточно продуманными именами предполагают плохо спроектированные API. Классы с широкими именами предполагают широкие API. Имя `Manager` — одновременно и непродуманное, и широкое. Слишком широкий API ведет к потере смысла и тормозит разработку: всякий раз, когда возникает необходимость взаимодействий с `TheBlob`, приходится анализировать

интерфейс, простирающийся на несколько экранов, и если не повезет, то еще и множество страниц документации.

Существует несколько приемов, помогающих минимизировать область видимости класса. Некоторые из них выражены в виде рекомендаций в Core Guidelines. Рекомендации «C.45. Не определяйте конструктор по умолчанию, который просто инициализирует переменные-члены; для этой цели лучше использовать внутриклассовые инициализаторы членов» и «C.131. Избегайте тривиальных геттеров и сеттеров», обсуждавшиеся в первой части, имеют побочный эффект. Они способствуют минимизации области видимости класса. Следуя рекомендации C.45, вы получаете на одну функцию меньше, а значит, меньшую область видимости, поскольку определение класса получается меньше на одно определение функции-члена, которое тоже является частью области видимости класса. К рекомендации C.131 применимо то же рассуждение: меньшее количество функций-членов подразумевает меньшую область видимости.

Рекомендация «C.4. Преобразуйте функцию в функцию-член, только если ей нужен прямой доступ к представлению класса» уменьшает область действия, заменяя функции-члены функциями, не являющимися ни членами, ни дружественными. Эти функции могут быть объявлены рядом с классом, но они не входят в область видимости класса. Конечно, это увеличивает размер области видимости пространства имен, но любое объявление неизбежно повлияет на ту или иную область видимости.

Простой отказ от расширения интерфейса класса выше определенного размера поможет сохранить области видимости классов небольшими. Можно даже решить пересматривать абстракцию всякий раз, когда количество общедоступных функций станет больше десяти, — возможно, ее получится разделить надвое. Может быть, получится разделить инварианты класса на две части и вывести более совершенную пару абстракций.

Тот же подход можно применить к уменьшению области видимости The Blob. Проверка инвариантов (часто класс начинается с нескольких инвариантов, количество которых растет со временем), их перечисление и классификация помогают выявить множество абстракций, моделирующее широкий набор содержащихся в нем понятий.

Мы надеемся, что это совпадение принципов наглядно показывает, насколько важна рекомендация, предлагающая минимизировать области видимости, следование ей вознаградится многократно.

ОБЛАСТЬ ВИДИМОСТИ ПАРАМЕТРОВ ФУНКЦИИ

Нам осталось обсудить три области видимости. Область видимости параметров функции немного похожа на область видимости блока, она отличается лишь добавлением сигнатуры функции. Область видимости параметров начинается с точки их объявления в сигнатуре функции и заканчивается в конце объявления или, если функция определяется, в конце тела функции. Например:

```
float divide(float a, // Область видимости начинается здесь
            float b); // Область видимости заканчивается здесь

float divide(float a, // Область видимости начинается здесь
            float b) {
    return a / b;
} // Область видимости заканчивается здесь
```

Есть еще один вариант, связанный с областью видимости функции, — блок `try`:

```
float divide(float a, // Область видимости начинается здесь
            float b)
try {
    std::cout << "Dividing\n";
    return a / b;
} catch (...) { // Область видимости продолжается здесь
    std::cout << "Dividing failed, was the denominator zero?\n";
} // Область видимости заканчивается здесь
```

Область видимости параметров функции заканчивается в конце последнего предложения `catch`. Она напоминает область видимости блока, чего и следовало ожидать, потому что функция подобна большому составному оператору. Вот еще одно совпадение рекомендаций в Core Guidelines: «F.3. Функции должны быть короткими и простыми». Следование этой рекомендации способствует минимизации области видимости функции. На самом деле соблюдение рекомендации «F.2. Функция должна выполнять одну логическую операцию» обычно приводит к короткой функции с небольшими областями видимости.

Классы с недостаточно продуманными именами предполагают плохо спроектированные API.

ОБЛАСТЬ ВИДИМОСТИ ПЕРЕЧИСЛЕНИЯ

Сущность области видимости перечисления кажется очевидной, и рекомендация минимизировать ее выглядит нелогичной. Например, существует 56 двухбуквенных аббревиатур, соответствующих названиям штатов США, 118 известных химических элементов и 206 костей в теле взрослого человека. Это константы, и они не зависят ни от каких рекомендаций минимизировать области видимости.

Однако это еще не все. Взгляните на следующие области:

```
enum US_state_abbreviation { // перечисление с неограниченной
                                // областью видимости
    ...
    VT,
    VA,
    VI,
    WA,
    WV,
    WI,
    WY
}; // Область видимости VI (Virgin Islands – Британские Виргинские
    // острова) не заканчивается здесь.

enum class Element { // перечисление с ограниченной областью видимости
    ...
    Nh,
    F1,
    Mc,
    Lv,
    Ts,
    Og
}; // Область видимости Lv (Livermorium – Ливерморий)
    // заканчивается здесь.

US_state_abbreviation southernmost = HI; // HI внутри области видимости
// Element_lightweight = H; // H вне области видимости
Element_lightweight = Element::H; // H внутри области видимости
```

Несмотря на то что область видимости `H` прерывается, вы все равно можете сослаться на этот член перечисления, явно указав область. Сделать это можно, как вы уже догадались, с помощью оператора разрешения области видимости `::`, записав ссылку как `Element::H`. Оператор разрешения области видимости сообщает компилятору: «Я имею в виду вот этот идентификатор, посмотри!»

Возможно, имя `Element` само находится в некотором пространстве имен. На самом деле мы надеемся, что это действительно так, потому что объявление чего-либо, кроме пространств имен, в глобальной области видимости — плохая идея. В этом случае мы можем сообщить компилятору пространство имен, в котором определено имя `Element`, например, так: `Chemistry::Element::H`. Можно провести параллели между областями видимости и глобальной системой доменных имен, где домены верхнего уровня, такие как `.com`, `.net` и коды стран, определяют области видимости; например, сравните `google.ie` и `google.fr`.

В примере выше у нас есть еще одно совпадение с рекомендациями из Core Guidelines. Рекомендация «Enum.3. Страйтесь использовать классы-перечисления вместо простых перечислений» обусловлена легкостью, с какой члены простых перечислений преобразуются в целые числа, а это приводит к некоторым нежелательным последствиям, описанным в предыдущей главе. Классы-перечисления, более известные как перечисления с ограниченной областью действия, сужают область видимости членов перечисления до границ его определения. Это помогает минимизировать область видимости, что особенно удобно для односимвольных идентификаторов, таких как `H`, обозначающий водород.

ОБЛАСТЬ ДЕЙСТВИЯ ПАРАМЕТРА ШАБЛОНА

Для полноты картины рассмотрим также область видимости параметров шаблона. Она начинается с точки объявления имени параметра и заканчивается в конце наименьшего объявления шаблона, в котором оно представлено. Например:

```
template< typename T, // начало области видимости Т
          T* p > // Т остается в области видимости
class X : public std::pair<T, T> // Т остается в области видимости
{
    ...
    T m_instance; // Т остается в области видимости
    ...
}; // конец области видимости Т
```

Это еще одна область видимости, размер которой зависит от внешних факторов, таких как размер определяемого класса. Минимизация этой области видимости невозможна без минимизации области видимости класса, поэтому тут мало что можно добавить.

ОБЛАСТЬ ВИДИМОСТИ КАК КОНТЕКСТ

Как видите, усилия по минимизации области видимости окупятся сторицей. По сути, области видимости отражают наши представления о делении вещей. Понятия области видимости и релевантности тесно связаны. Программный код можно представить как повесть, состоящую из глав, каждая из которых представляет отдельную область видимости. Все объявления в коде отображаются в некоторую область видимости (желательно не в глобальную), и очевидно обоснованным шагом является объединение родственных объявлений в одну область и точное указание ассоциаций в минимально возможных областях.

Области видимости заключают и идентифицируют абстракции. Области видимости — это наборы имен, будь то область видимости класса, функции или пространства имен. Они содержат объявления, относящиеся к текущей абстракции, являются основными строительными блоками области решений. Если области видимости будут небольшими, то абстракции тоже будут небольшими.

Однако не все области видимости имеют имена, и не всем абстракциям нужны имена. Вполне допустимо открыть область видимости блока внутри другой области видимости и использовать ее для решения некоторой небольшой автономной задачи. Возможно, ее потребуется ограничить с помощью `std::scoped_lock`. Однако имейте в виду, что такие вложенные области видимости несут риск сокрытия имен. Вы можете непреднамеренно прервать область видимости существующего имени, повторно объявив ее во вложенной области.

Область видимости и продолжительность хранения взаимосвязаны, но не всегда взаимозаменяемы. Например, все имена в глобальном пространстве имен относятся к статическому классу хранения. Однако имена с областью видимости перечисления вообще не связаны с объектами; они являются простыми символическими константами, не требующими хранения.

Перемещение внимания от области к области требует ментального переключения контекста так же, как чтение книги требует от читателя построения модели сюжета и персонажей. Область видимости можно рассматривать как непосредственный контекст происходящего в программе. Он делит область решения на части, более простые для восприятия. Поняв взаимосвязь между областью видимости, контекстом и абстракцией, вы сможете делить любые задачи на управляемые фрагменты и получать удовольствие от программирования.

ПОДВЕДЕМ ИТОГ

- Области видимости имеют имена, а не объекты.
- Различайте области видимости и классы хранения.
- Удобочитаемость кода обратно пропорциональна размеру области видимости.
- Минимизируйте области видимости, чтобы свести к минимуму время удержания ресурсов объектами с автоматическим классом хранения.
- Опасайтесь сокрытия имен при вложении областей видимости друг в друга.
- Страйтесь использовать перечисления с ограниченной областью видимости, чтобы минимизировать область видимости их членов.
- Интерфейсы должны быть минимальными и полными, чтобы минимизировать области видимости.
- Стремитесь минимизировать области видимости, чтобы оптимизировать абстракции.

Области видимости
заключают и иденти-
фицируют абстракции.

ГЛАВА 4.4

Con.5. Используйте `constexpr` для определения значений, которые можно вычислить на этапе компиляции

ОТ `CONST` К `CONSTEXPR`

До появления C++11 механизм `const` ограничивался только квалификацией типа как `const`, а следовательно, любого экземпляра этого типа как неизменяемого и квалификацией нестатической функции-члена, чтобы `*this` в ее теле интерпретировался как константа, например:

```
class int_wrapper {
public:
    explicit int_wrapper(int i);
    void mutate(int i);
    int inspect() const;

private:
    int m_i;
};

auto const i = int_wrapper{7}; // i получает тип int_wrapper const
// i.mutate(5); // Нельзя вызвать неконстантную функцию-член
                  // для константного объекта
auto j = i.inspect(); // Присваивание значения, возвращаемого inspect
```

Мы уверены, что вы знакомы с этими вариантами применения `const`. Возможно, вы также знакомы с ключевым словом `mutable`, квалифицирующим нестатические переменные-члены как невосприимчивые к ограничениям `const`, особенно если вы читали главу 3.4 «ES.50. Не приводите переменные с квалификатором `const` к неконстантному типу». Это часть механизма `const` в той мере, в какой она применяется к функциям-членам с квалификатором `const`.

В C++11 появилось новое ключевое слово `constexpr`. Идея его появления заключалась в том, чтобы позволить выполнять во время компиляции очень ограниченный набор функций. До этого инженеры уже поступали так, используя макросы препроцессора. С появлением `constexpr` стало возможным отказаться от этих макросов и заменить их точными типобезопасными функциями. Было приятно чувствовать возможность исключить еще один вариант использования препроцессора.

Однако не обошлось без ограничений: в функциях с квалификатором `constexpr` разрешалось использовать только одно выражение, что привело к возрождению рекурсии и широкому использованию тернарных операторов. Стандартным примером метапрограммирования шаблонов было создание функций вычисления факториалов и чисел Фибоначчи, но также имелась возможность реализовать подобным образом множество математических функций, таких как тригонометрические расширения. Возможность применения только одного выражения сильно ограничивала, но оттавивала всеобщее представление о функциональном программировании.

Вот пример вычисления факториала:

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 // Терминальная ветвь
                  : (n * factorial(n - 1)); // Рекурсивная ветвь
}
```

Рекурсия наглядно показала всем ограничения компилятора: $12!$ является наибольшим факториалом для 32-битного представления.

Этот прием приобрел фантастическую популярность, а позже C++14 еще больше расширил его область применения. Ограничение одним выражением было снято, и начали появляться целые библиотеки `constexpr`. В частности, стало возможным хеширование во время компиляции. Так джин был выпущен из бутылки, и ничто не могло остановить распространение `constexpr` по всему языку и ослабление ограничений.

Теперь можно написать функцию вычисления факториала, используя старые добрые операторы `if-then`:

```
constexpr int factorial(int n) {
    if (n <= 1) return 1; // Терминальная ветвь
    return n * factorial(n - 1); // Рекурсивная ветвь
}
```

C++17 принес еще больше инноваций: появилась возможность объявлять лямбда-выражения как константные выражения `constexpr`. Это может пока-

заться странным, но если подумать о лямбда-выражении как о синтаксическом сахаре, определяющем класс с оператором вызова функции, то лямбда-выражение с квалификатором `constexpr` можно интерпретировать как оператор вызова функции `constexpr`. Кроме того, была добавлена новая конструкция: `if constexpr`. Она позволяет устранить некоторые случаи использования `std::enable_if` и уменьшить количество перегруженных версий шаблонов функций. Это еще больше упростило код, сократило время компиляции и облегчило понимание кода за счет устраниния части SFINAE.

В C++20 появилось то, что десять лет назад показалось бы действительно странным: виртуальные функции `constexpr`, блоки `try/catch` в функциях `constexpr`, алгоритмы `constexpr` в STL и динамическое распределение `constexpr`, ведущее к `constexpr std::string` и `constexpr std::vector`. Да, так много в настоящее время стало доступно во время компиляции. Можно создавать обширные библиотеки, полностью основанные на `constexpr`. Хана Душикова (Hana Dusíková) удивила сообщество на CppCon в 2017 году библиотекой регулярных выражений времени компиляции¹.

Средства `constexpr` превратились в язык внутри языка, заменяя препроцессор во многих местах типобезопасными вариантами, учитывающими область видимости². Эта рекомендация была впервые предложена еще во времена C++14, но сейчас стадия роста ее области применения, можно сказать, расширилась до того, что скоро будет выражаться в тезисе «используйте `constexpr` везде, где это возможно» или даже «используйте `constexpr` по умолчанию».

C++ ПО УМОЛЧАНИЮ

C++ имеет репутацию языка, неправильно определяющего значения по умолчанию. Например, нестатические функции-члены по умолчанию являются изменяемыми, тогда как безопаснее было бы сделать их константными. Изменение состояния требует больше размышлений, рассуждений и опасений, чем проверка состояния, поэтому создание изменяемой нестатической функции-члена должно быть осознанным извешенным решением. Инженер должен сформулировать: «Я решил сделать эту функцию изменяемой, и я явно заявляю об этом». В соответствии со стандартом нестатическая

¹ https://www.youtube.com/watch?v=3WGaN_Hp9QY

² Reis G. D., Stroustrup B. General Constant Expressions for System Programming Languages. SAC-2010. The 25th ACM Symposium on Applied Computing. March 2010.

функция-член может вести себя так, как если бы она была дополнена квалификатором `const`, но если инженер забудет применить этот квалификатор, то эта информация не будет передана клиенту. Запрет на изменение, принятый по умолчанию, оказывается безопаснее, чем разрешение.

Средства `constexpr` превратились в язык внутри языка, заменяя препроцессор во многих местах типобезопасными вариантами, учитывающими область видимости.

Эти рассуждения можно распространить и на типы. По умолчанию экземпляр типа является изменяемым. Он становится неизменяемым, только если его тип объявлен константным. Опять же безопаснее предотвратить изменение чего-либо без явного разрешения, чем разрешить свободное изменение. Если бы такое было реализовано, без квалификатора объект мог бы вести себя так, как если бы он был константным, и в целом было бы яснее, если бы все было константным, когда не указано иное.

Атрибут `[[no_discard]]` — еще один кандидат на операцию по умолчанию. Вот как это выглядит на деле:

```
[[no_discard]] bool is_empty(std::string const& s)
{
    return s.size() == 0;
}
```

Когда возвращаемые значения, содержащие коды ошибок, можно просто отбросить, это позволяет игнорировать ошибки. Вызывающий код должен как минимум подтвердить, что он игнорирует ошибку. То же касается операций префиксного и постфиксного инкремента. Префиксный инкремент увеличивает указанный объект и возвращает его. Операция постфиксного инкремента создает копию объекта, увеличивает объект и возвращает копию. Если используется постфиксный инкремент, а возвращаемая копия отбрасывается, становится очевидно, что сохранение копии было напрасной тратой усилий. Здесь не поможет и правило «как если бы»: если компилятор не знает об отсутствии наблюдаемой разницы, например, в случае встроенного типа, то он не сможет заменить постфиксный инкремент префиксным. Атрибут `[[no_discard]]` сигнализировал бы о том, что код выполняется впустую. Наконец, функция `empty()` возвращает логическое значение, и ее часто путают с `clear()`. Атрибут `[[no_discard]]` будет сигнализировать о том, что возвращаемое значение функции, которая ничего не делает с объектом, игнорируется и вызов фактически является излишним.

Однако давайте вспомним историю C++. Этот язык разрабатывался как расширение С, и в свое время было принято решение гарантировать, что компилятор C++ будет по-прежнему компилировать программы на С. Если бы эти значения по умолчанию поменялись местами, то все программы на языке С перестали бы компилироваться. Успех C++ во многом обусловлен его совместимостью с кодом, созданным в предыдущих версиях языка. Создать новый язык просто. Для некоторых инженеров это хобби. Но заставить других инженеров использовать его очень сложно, и, конечно же, требование полностью переписать существующий код оказалось бы слишком высокой ценой за переход на новую версию.

ИСПОЛЬЗОВАНИЕ CONSTEXPR

Как было показано в примерах, ключевое слово `constexpr` простое в использовании. Достаточно снабдить функцию ключевым словом `constexpr`, и все готово. Однако взгляните на следующую функцию:

```
int read_number_from_stdin()
{
    int val;
    cin >> val; // Что даст эта инструкция во время компиляции?
    return val;
}
```

Ясно, что эта функция как `constexpr` не имеет смысла, потому что ее часть — взаимодействие с пользователем — возможна только во время выполнения.

Вы можете спросить: «Почему вся стандартная библиотека не переписана с применением `constexpr`?»

Тому есть две основные причины. Во-первых, не все функции в стандартной библиотеке можно объявить как `constexpr`. Например, доступ к файловому потоку зависит от клиентской файловой системы, и декорирование `std::fopen` как `constexpr` в большинстве ситуаций бессмысленно, так же как и функции, получающей ввод из `stdin`. Такие функции, как `std::uninitialized_copy`, тоже относятся к категории проблемных в отношении `constexpr`. Что может означать неинициализированная память в контексте `constexpr`?

Во-вторых, такое предложение еще не было сделано. Чтобы выдвинуть на обсуждение документ под названием «Переделать стандартную библиотеку

под `constexpr»`, потребовалось бы внести огромное количество изменений в формулировки стандарта. C++20 отводит стандартной библиотеке 1161 страницу из 1840. Если такой документ и появится, то он будет отклонен еще до серьезного рассмотрения на заседании комитета. Кроме того, даже просто выяснить, какие функции можно объявить `constexpr`, — гигантская по своим масштабам задача. Любая функция, вызываемая функцией `constexpr`, тоже должна быть `constexpr`, а это означает, что вам придется последовательно проверить возможность применения `constexpr` ко всем вызываемым функциям, пока не встретится промежуточная или конечная функция, которую нельзя объявить как `constexpr`, и вы вынуждены будете вернуться к тому, с чего начали. Строить такое дерево проверок функций — не самое увлекательное занятие. Когда механизм `const` только начал становиться серьезным инструментом программирования, часто приходилось гоняться за `const` по графу вызовов и выбирать функции, которые не могут быть константными. Встав на этот путь, вы не сможете остановиться, потому что `const` и `constexpr` привязываются как вирусы, распространяясь через все, к чему они прикасаются, в том и заключаются их красота и ужас.

Однако среди кандидатов на объявление `constexpr` все еще остаются контейнеры, которые действительно могут быть `constexpr`. Например, без особого труда можно реализовать свои типы `map` и `unordered_map` как `constexpr`, с идентичными для типов `std::map` и `std::unordered_map` API. А с помощью `std::vector`, `std::string` и типа `constexpr map`, упорядоченного или неупорядоченного, можно реализовать несколько очень полезных парсеров, действующих во время компиляции, и использовать их для настройки сборок.

Рассмотрим упрощенный пример. Пусть требуется вычислить $\sin x$, используя ряд Тейлора:

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

Для этого нужны только две функции — степенная функция и функция вычисления факториала. Суммирование членов ряда сделает все остальное.

У нас уже есть функция факториала. Добавим степенную функцию:

```
constexpr double pow(double t, unsigned int power) {
    if (power == 0) return 1.0;
    return t * pow(t, power - 1);
}
```

Теперь можно добавить функцию вычисления синуса `sin`:

```
constexpr double sin(double theta, int term_count) {
    auto result = 0.0;
    while (term_count >= 0) {
        auto sign = pow(-1, (term_count));
        auto numerator = pow(theta, (2 * term_count) + 1);
        auto denominator = factorial((2 * term_count) + 1);
        result += (sign * numerator) / denominator;
        --term_count;
    }
    return result;
}
```

Значение `term_count` определяет запрашиваемую точность. Каждый последующий член меньше предыдущего, и в конечном итоге вклад очередного члена будет пренебрежимо мал. Есть еще одна причина для введения этого параметра, к ней мы вскоре вернемся. Теперь можно завершить пример функцией `main`:

```
#include <numbers>

int main()
{
    return sin(std::numbers::pi / 4.0, 5) * 10000;
}
```

Здесь для демонстрации мы вызываем функцию `sin` и преобразуем результат в целое число, чтобы его можно было вернуть.

Если ввести этот код на странице Compiler Explorer, то станет заметно, что компилятор генерировал всего две ассемблерные инструкции:

```
mov eax,7071
ret
```

$\sin \pi/4$ в радианах равен $1/\sqrt{2}$, что примерно равно 0,7071, то есть можно быть уверенными в правильности этой функции. Если вас интересует точность, то попробуйте увеличить скалярный множитель до 1 000 000 или количество членов в ряду Тейлора.

Увеличение количества членов — вот где кроется самое интересное. На момент написания этих строк при увеличении количества членов до шести компилятор gcc генерировал код для функции `pow`. Функция `pow` больше не поддается оценке во время компиляции, и вычисления откладываются до времени выполнения. Функция `pow` должна вычислить $\pi/4$ в 13-й степени, что, по-видимому, является точкой, в которой компилятор решает, что мы

требуем от него слишком много. Увеличьте количество членов до восьми, и компилятор также будет генерировать код для функции факториала.

Другая полезная особенность `constexpr` — запрет неопределенного поведения и побочных эффектов в функции. Например, рассмотрим следующий опасный фрагмент кода:

```
int get_element(const int *p) {
    return *(p + 12);
}

int main() {
    int arr[10]{};
    int x = get_element(arr);
}
```

Здесь проблема очевидна: функция `get_element` извлекает значение 12-го элемента массива, тогда как ей передается массив всего с десятью элементами. Это неопределенное поведение. Представьте, что случится, если объявить константными выражениями `constexpr` все (кроме `main()`, которая не может быть константным выражением):

```
constexpr int get_element(const int *p) {
    return *(p + 12);
}

int main() {
    constexpr int a[10]{};
    constexpr int x = get_element(a);
    return x;
}
```

В этом случае компилятор сообщит об ошибке:

```
example.cpp
<source>(8):
error C2131: expression did not evaluate to a constant1
<source>(2):
note: failure was caused by out of range index 12;
      allowed range is 0 <= index < 102
Compiler returned: 2
```

Запрет неопределенного поведения в функциях `constexpr` помогает выяснить при компиляции места, где вы этим пользуетесь. Этот выбор должен

¹ Выражение нельзя вычислить как константу. — *Примеч. пер.*

² Ошибка вызвана выходом индекса 12 за границы диапазона; допустимый диапазон $0 \leq \text{индекс} < 10$. — *Примеч. пер.*

быть сознательным, и вам решать, является ли использование неопределенного поведения той ценой, которую вы готовы заплатить. Так или иначе, используя `constexpr` для этих целей, будьте очень осторожны. Следующий код вполне успешно компилируется:

```
constexpr int get_element(const int *p) {
    return *(p + 12);
}

int main() {
    constexpr int a[10]{};
    return get_element(a);
}
```

Поэкспериментируйте с кодом этой главы на сайте Compiler Explorer.

Запрет побочных эффектов в функциях `constexpr` превращает их в чистые функции. Их легче тестировать, а результаты можно кэшировать, но самое замечательное — их можно вызывать параллельно без синхронизации. Синхронизация — это тяжелое бремя, от которого стоит избавиться.

Теперь самое время поговорить о `constexpr`. Но сначала поговорим об `inline`.

INLINE

`inline` — старое, очень старое ключевое слово. Оно появилось еще в языке C. В C++ этот спецификатор может применяться к функции или объекту и означает, что определение можно найти в единице трансляции. Он также сообщает компилятору, что функция является подходящим кандидатом для подстановки ее тела в точку вызова. Это свойство делает его целью для оптимизации: зачем платить за вызов функции, если можно просто подставить код?

На самом деле есть несколько причин, по которым такое избавление от функции нежелательно.

Когда вызывается функция, сохраняется некоторый контекст процессора, чтобы вызывающая функция могла продолжить работу после возврата. Для этого значения регистров обычно помещаются в стек, а затем восстанавливаются после завершения функции. Кто должен сохранять и восстанавливать регистры — вызывающая или вызываемая функция, определяется соглашениями о вызовах. Дополнительные инструкции, обеспечивающие сохранение и восстановление контекста, влияют на размер и скорость выполнения программы. В идеале подстановка положительно влияет и на то

и на другое. Однако если встраиваемая функция чересчур велика, то подстановка может привести к увеличению размера программы. Тем не менее иногда встраивание функций дает положительный эффект, увеличивая скорость выполнения кода за счет отказа от сохранения регистров.

В прошлом веке это было вполне разумное решение. Однако современные процессоры имеют кэши инструкций. Если сделать подстановку, есть риск заполнить кэш инструкций и получить промах кэша. Если вызывающая функция выполняет цикл, а встраиваемая функция вызывается лишь изредка, то имеет смысл вынести ее тело за рамки цикла. Компилятор знает гораздо больше о характеристиках процессора, чем вы, поэтому решение лучше оставить за компилятором. Кроме того, правило «как если бы» позволяет компилятору выполнить подстановку любой функции, если это поможет повысить производительность. В конце концов, весь смысл функций в том, что они являются невидимыми и удобными фрагментами кода, который расположен «где-то еще». Их расположение не имеет значения. Они могут существовать как в точке вызова, так и далеко от нее. В свете этого можно заключить, что ключевое слово `inline` в значительной степени избыточно. Это подсказка компилятору, что функция является кандидатом на подстановку, но не обязательное требование замены функции подстановкой. На самом деле единственная ситуация, в которой подстановка действительно произойдет, — когда отмеченная компилятором функция появится более чем в одной единице трансляции и если функция определена в заголовочном файле.

Некоторые реализации компиляторов предлагают нестандартный спецификатор, такой как `_forceinline`, означающий ваше мнение: «Это мое решение. Я прекрасно осознаю, что могу выстрелить себе в ногу. Я считаю, что обладаю всеми необходимыми знаниями, которых нет у тебя, компилятор». Как правило, если функция не может быть встроена, компилятор выдает ошибку и сообщает, что на самом деле вы сильно ошибаетесь.

Важно помнить, что `inline` — это подсказка, а `_forceinline`, если поддерживается реализацией, — это команда.

CONSTEVAL

Как вы наверняка заметили, экспериментируя с функцией `constexpr sin`, можно достаточно сильно нагрузить компилятор, чтобы он сдался и отложил вычисления до этапа выполнения. Подобно `inline`, подсказке компилятору, что он может заменить вызов указанной функции ее телом,

ключевое слово `constexpr` тоже является подсказкой компилятору, что он может попытаться выполнить вычисления во время компиляции. Это необязательное требование, и если оно невыполнимо, то компилятор проигнорирует его молча. Ключевое слово `constexpr` означает только то, что данное выражение может быть оценено во время компиляции, но это не обязательно. Если в выражении используются значения, которые будут известны только во время выполнения, то компилятор просто отложит оценку до этапа выполнения.

Есть некоторые причины, почему нежелательно, чтобы это произошло. У вас может быть что-то, что может выйти из-под контроля при определенных обстоятельствах; например, итеративная система функций, которая никогда не стабилизируется. Было бы проще положиться на компилятор, который будет вызывать функцию, только когда это можно сделать безопасно.

Это цель ключевого слова `consteval`. Оно указывает, что функция является непосредственной, то есть каждый вызов функции должен производить значение, вычисляемое во время компиляции. Это аналог `_forceinline`, компилятор сообщит об ошибке, если указанная функция не может создать такое значение.

Ключевое слово `consteval` требует осторожного обращения. Оно не гарантирует обязательную оценку выражения во время компиляции. А пользователь должен ограничить входные данные, чтобы не «сжечь» компилятор. Это требование не переносимо в другие среды, и фактическое проявление таких ограничений может быть неочевидно для пользователя. Экспериментировавшие с функцией `sin` в Compiler Explorer могли заметить, что некоторые реализации никогда не вычисляют результат во время компиляции. Если отметить такие функции ключевым словом `consteval`, это доставит массу неудобств вашим пользователям. Для начала можно отметить функции ключевым словом `constexpr`, а затем, если в некоторых случаях оценка, которую хотелось бы выполнить во время компиляции, откладывается до этапа выполнения, можно заменить `constexpr` на `consteval`.

CONSTINIT

В завершение этой главы мы поговорим о ключевом слове `constinit`. Но прежде вы должны получить четкое представление о нулевой инициализации и константной инициализации. К сожалению, инициализация — огромная тема, достойная отдельной книги, поэтому мы сосредоточимся только на этих двух аспектах инициализации в C++.

Нулевая инициализация — прекрасный пример выбора хорошего имени. Под нулевой инициализацией подразумевается присваивание объекту начального нулевого значения. В C++ есть небольшая проблема, заключающаяся в отсутствии специального синтаксиса нулевой инициализации. Она выполняется в нескольких случаях, например:

```
static int j;
```

Обычно, если объект объявляется без инициализации, его начальное значение оказывается не определено, если нет конструктора по умолчанию. Поскольку тип `int` не имеет конструктора, можно ожидать, что `j` получит случайное значение. Однако существует специальное исключение для объектов со статическим классом хранения: они инициализируются нулями, как на этапе статической инициализации перед вызовом `main()`. В частности, объект получит нулевое значение, явно преобразованное в соответствующий тип.

Если объект имеет тип класса, то все базовые классы и нестатические переменные-члены инициализируются нулями. Конструкторы при этом игнорируются, что стоит иметь в виду. Некоторые реализации поддерживают инициализацию по умолчанию неинициализированных объектов с использованием параметров командной строки. Это полезно для отладки, поскольку в таком случае можно задать в качестве значения по умолчанию необычный битовый шаблон и по нему обнаружить неинициализированные объекты с помощью инструментов проверки памяти.

Константная инициализация устанавливает начальное значение статических переменных равным константе времени компиляции. Если у типа есть конструктор `constexpr` и все аргументы являются константными выражениями, то переменная инициализируется константой. Если тип не имеет конструктора, или экземпляр инициализируется явно заданным значением, или инициализатор является константным выражением, то переменная также инициализируется константой. Например:

```
const int j = 10;
```

инициализируется константой, если объявление находится на уровне области видимости файла и, соответственно, имеет статический класс хранения. Константная инициализация выполняется до статической инициализации. На практике это происходит во время компиляции.

Ключевое слово `constinit` сообщает, что объект имеет статическую инициализацию, и вызывает сбой компиляции, если это не так. Например:

```
constinit const int j = 10;
```

Такой вариант использования несколько избыточный, потому что это выражение не противоречит утверждению.

Более интересный пример:

```
// definition.h
constexpr int sample_size() {
    // определение функции
}

// client.cpp
constinit const int sample_total = sample_size() * 1350;
```

Встретив это выражение, компилятор сообщит об ошибке, если автор `sample_size()` изменит функцию так, что ее нельзя будет вычислить на этапе компиляции. Это ценная особенность. Объекты, отмеченные как `constinit`, оцениваются во время компиляции, а не во время выполнения, что помогает смягчить проблему фиаско порядка статической инициализации.

ПОДВЕДЕМ ИТОГ

Использование ключевых слов с префиксом `const` позволяет предложить компилятору выполнить вычисления на этапе компиляции. Вычисления на этапе компиляции предпочтительнее вычислений на этапе выполнения: это выгоднее для клиента.

- Используйте `constexpr`, чтобы подсказать компилятору, что значение функции можно вычислить на этапе компиляции, и обнаружить неопределенное поведение, если оно имеет место.
- Используйте `consteval`, чтобы убедить компилятор в том, что функцию можно вычислить на этапе компиляции, хотя бы и с риском получить сообщение об ошибке в ответ.
- Используйте `constinit`, чтобы гарантировать инициализацию объекта на этапе компиляции.

ГЛАВА 4.5

Т.1. Используйте шаблоны для повышения уровня абстрактности кода

В 2012 году мы с коллегой, назовем ее Бет, провели проверку кода, который считаю полезным обсудить сейчас¹. Цель проверяемой функции состояла в том, чтобы получить набор значков, расположенных в ряд вдоль нижней границы экрана, и прокрутить их на произвольную величину. В ту пору мы только что перешли на C++11.

После получаса наших трудов функция потеряла около десятка ненужных строк кода и стала выглядеть так:

```
UIComponent* spin_cards(UIComponent* first,
                        UIComponent* n_first,
                        UIComponent* last) {
    if (first == n_first) return last;
    if (n_first == last) return first;
    UIComponent* write = first;
    UIComponent* next_read = first;
    UIComponent* read = n_first;
    while (read != last) {
        if (write == next_read) {
            next_read = read;
        }
        std::iter_swap(write++, read++);
    }
    spin_cards(write, next_read, last);
    return write;
}
```

В этот момент я задал Бет три вопроса. Первый: «Вы думали об использовании `auto` для `write`, `next_read` и `read`?» Она закатила глаза и спросила:

¹ Рассказывает Дж. Гай Дэвидсон, и опять на протяжении целой главы.

«Зачем?» Потом мы поговорили о программировании с использованием интерфейсов вместо типов. В конце я попросил ее порадовать меня и внести изменения в код:

```
UIComponent* spin_cards(UIComponent* first,
                        UIComponent* n_first,
                        UIComponent* last) {
    if (first == n_first) return last;
    if (n_first == last) return first;
    auto write = first;
    auto next_read = first;
    auto read = n_first;
    while (read != last) {
        if (write == next_read) {
            next_read = read;
        }
        std::iter_swap(write++, read++);
    }
    spin_cards(write, next_read, last);
    return write;
}
```

Затем я спросил: «Какую часть интерфейса `UIComponent` вы используете в этой функции?» И она внимательно посмотрела на меня. Попутно мы заменили ручной обмен вызовом `std::swap`, затем заменили этот вызов на `std::iter_swap`. Я ждал, когда зазвенит звоночек и Бет поймет, что `UIComponent` — не что иное, как прямой итератор, но я должен был подвести ее к этому моменту. После некоторого мычания и оханья она ответила: «Вообще-то, я могла бы и сразу создать шаблон для прямого итератора. Все, что я делала до этого момента, — присваивала, сравнивала, разыменовывала и записывала». Тридцать секунд спустя мы имели:

```
template <typename T>
T* spin_cards(T* first, T* n_first, T* last) {
    if (first == n_first) return last;
    if (n_first == last) return first;
    auto write = first;
    auto next_read = first;
    auto read = n_first;
    while (read != last) {
        if (write == next_read) {
            next_read = read;
        }
        std::iter_swap(write++, read++);
    }
    spin_cards(write, next_read, last);
    return write;
}
```

Я предложил ей выбрать менее конкретное и более описательное имя. Пока она размышляла о сложности именования, я спросил ее, сколько времени у нее ушло на написание и тестирование функции. «Около двух часов, — ответила она. — Я не сразу разобралась в некоторых аспектах, и это задержало меня».

«Хм-м-м, — сказал я. — Возможно, вы могли бы дать этой функции имя `rotate`». Она посмотрела на меня недоверчиво и несколько подавленно, когда поняла, что краткое описание ее разработки напоминает `std::rotate`. Мы сравнили наш последний вариант с определением реализации `std::rotate` и обнаружили, что они очень и очень похожи. Описание `std::rotate` вы найдете по адресу <https://ru.cppreference.com/w/cpp/algorithm/rotate>.

ПОВЫШЕНИЕ УРОВНЯ АБСТРАКЦИИ

Моя любимая часть разработки — удаление лишнего кода. Чем меньше кода, тем ниже когнитивная нагрузка и тем проще понять код. Цель инженера должна состоять в том, чтобы выразить область решения как можно проще и понятнее. Платить инженерам «за строки кода» — верный способ уничтожить проект.

Рассмотрим разницу между тем, что компилятор получает на входе, и тем, что он выдает на выходе. Обычно ему передается единица трансляции с сотнями строк кода, а на выходе получаются тысячи ассемблерных инструкций, заключенных в объектный файл. Каждая из этих ассемблерных инструкций заставляет процессор выполнить некоторые операции. Ассемблерный код сложнее и подробнее кода на C++. Процессорные операции сложнее, чем ассемблерные инструкции. Все усилия разработчиков программного обеспечения и языков программирования направлены на снижение сложности и детализации.

Когда я учился своей профессии в 1980-х, меня знакомили с концепцией языков высокого уровня, таких как Sinclair BASIC и BBC BASIC, и языков низкого уровня, таких как ассемблер Z80 и ассемблер 6502. Под уровнем в данном случае понимается уровень абстракции. Дистанция между BASIC и ассемблером значительная.

Когда я впервые начал изучать C, уже будучи знакомым с BASIC и ассемблером, нам представили его как низкоуровневый язык, хотя и не настолько низкоуровневый, как ассемблер. Я воспринимал C как язык

программирования макросов для ассемблера. С ним я мог видеть, какие ассемблерные инструкции будут генерированы разными конструкциями на С. Мне стала доступна непосредственность ассемблера в сочетании с ясностью именования BASIC. Я мог давать имена переменным, как в BASIC, и легко мог предположить, сколько памяти они займут. Я мог создавать структуры управления циклами без необходимости возиться с правильной проверкой флагов, чтобы решить, вернуться в начало цикла или продолжить.

Сложность ассемблера была уменьшена за счет использования более удобных идентификаторов и четкой структуры. Язык С поднял уровень абстракции.

Впервые столкнувшись с C++, я поначалу воспринял его как язык программирования макросов для С. Часть хлопотных операций, которые я писал, например, объединение `malloc` и `init` в одну функцию, выполнялись за меня компилятором C++ в форме новых ключевых слов. C++ еще выше поднял уровень абстракции.

В обоих случаях повышение уровня абстракции не привело к заметному снижению производительности кода. Сначала компиляторы были несовершенны, и иногда мне доводилось открывать рот, удивляясь кажущемуся идиотизму автора компилятора, но, к счастью, я мог вернуться на более низкий уровень абстракции и вставить свои ассемблерные инструкции. Я помню, как в последний раз попытался превзойти компилятор примерно в 1991 году. С тех пор я не раз убеждался, что он работает лучше меня, на него можно положиться.

Важно отметить, что, если бы мне пришлось вернуться к ассемблеру, я был бы уже не так хорош, как раньше, потому что сложность процессоров сильно возросла. Когда я писал код на ассемблере, процессоры имели одно ядро, не имели кэшей и, следовательно, не было ни предварительной выборки данных, ни прогнозирования переходов. Решение использовать непрерывные структуры данных вместо данных на основе узлов сыграло важную роль. Операции с памятью, чтение/запись, стали занимать примерно столько же времени, сколько цикл выполнения инструкции. Нотация «О большое» с очевидностью доказывала свою состоятельность.

Повышение уровня абстракции улучшает переносимость кода. Мой красивый код на ассемблере работал только на одном семействе процессоров. В начале 1980-х это было серьезной проблемой, потому что существовали

два конкурирующих домашних компьютера, считавшихся ведущими игровыми платформами, и они работали на процессорах Z80 и 6502. К середине 1980-х все упростилось благодаря выбору процессора 68000 как для Atari ST, так и для Commodore Amiga, а также для первых компьютеров Apple Mac.

Цель API — уменьшить сложность, скрыв детали за интерфейсом и повысив уровень абстракции. Цель языка — скрыть сложность индивидуального смысла и тонкостей реализации языковых компонентов. Повышение уровня абстракции имеет основополагающее значение для работы инженера-программиста.

Чем меньше кода, тем ниже когнитивная нагрузка и тем проще понять код. Цель инженера должна состоять в том, чтобы выразить область решения как можно проще и понятнее. Платить инженерам «за строки кода» — верный способ уничтожить проект.

ШАБЛОНЫ ФУНКЦИЙ И АБСТРАКЦИЯ

Надеюсь, я убедил вас в важности повышения уровня абстракции как части практики программирования. Давайте теперь рассмотрим некоторые примеры.

Прочитав историю в начале главы, вы могли заметить, что все началось с решения, нацеленного на конкретную задачу, а именно на изменение порядка набора значков. Во время обзора кода имел место важный момент, когда мы ввели `std::swap`. Код выглядел так:

```
UIComponent* spin_cards(UIComponent* first,
                        UIComponent* n_first,
                        UIComponent* last) {
    if (first == n_first) return last;
    if (n_first == last) return first;
    UIComponent* write = first;
    UIComponent* next_read = first;
    UIComponent* read = n_first;
    while (read != last) {
        if (write == next_read) {
            next_read = read;
        }
        UIComponent tmp = *write;
        *write++ = *read;
        *read++ = tmp;
    }
}
```

```
    spin_cards(write, next_read, last);
    return write;
}
```

Здесь видно, как вручную выполняется обмен значениями в конце цикла `while`. Я задал Бет вопрос: «Вы решили не использовать `std::swap`?» И она ответила: «Я не видела в этом смысла. `UIComponent` — тривиальный объект, и его замена выполняется просто, зачем вызывать функцию?»

Моей немедленной реакцией было упоминание «декларации о намерениях». Для этого в Core Guideline имеется рекомендация «Р.3. Выражайте свои намерения». Да, происходящее очевидно: мы вручную реализовали обмен значениями, использовав времененную переменную и т. д. Заменив затем эту последовательность операций одним словом `swap`, мы сделали код понятнее без дополнительных затрат. Даже самый простой компилятор заменит вызов функции в реализации. Кроме того, если `UIComponent` вдруг перестанет быть тривиальным объектом и потребует собственной специализации `swap`, код выиграет от этого изменения в деталях. Удаление этой маленькой детали и замена ее шаблоном функции позволили уменьшить сложность, повысило уровень абстракции, сделало код более выразительным и добавило уверенности в будущем.

Конечно, в стандартной библиотеке уже есть обильный источник полезных шаблонов функций, особенно в заголовке `<algorithm>`. Высока вероятность, что написанный вами цикл можно заменить одним из шаблонов функций, определенных в этом заголовке. Ну а если этого нельзя сделать, то вполне возможно, что вы нашли новый алгоритм. Вы должны поделиться им с миром.

Мы говорим это вполне серьезно.

Вероятно, в заголовке `<algorithm>` есть некоторые упущения. Например, в C++20 стандартизированы `std::shift_left` и `std::shift_right`. Они сдвигают элементы в диапазоне, я и сам довольно часто делал это своими кодами. Мне даже в голову не приходило, что другие люди могут заниматься чем-то настолько фундаментальным. Когда я увидел документ касательно этой стандартизации, предложенный комитету, я почувствовал себя довольно глупо, это был момент озарения: я осознал, насколько часто буду использовать эту стандартную функцию.

Некоторые шаблоны функций в заголовке `<algorithm>` используют другие функции из того же заголовка. Например, `std::minmax` будет вызывать `std::min` и `std::max`. Библиотека использует все возможности для повышения уровня абстракции, что приводит к повторному использованию кода и, следовательно, к уменьшению количества новых строк кода.

Как только вы начнете заменять свои циклы существующими шаблонами функций, ваш код станет намного более выразительным и менее подверженным ошибкам. Если вы невольно пытаетесь написать `std::find_if`, то замена вашей попытки проверенной и отлаженной версией, написанной людьми, понимающими в компиляции больше, чем вы, безусловно, будет правильным решением. Конечно, прежде, чем вы сможете это сделать, вы должны осознать, что действительно пытаетесь переписать одну из функций в `<algorithm>`. Лучшим индикатором этого является использование цикла.

Алгоритмов очень много. Посмотрите на следующую функцию:

```
bool all_matching(std::vector<int> const& v, int i) {
    for (auto it = std::begin(v); it != std::end(v); ++it){
        if (*it != i) {
            return false;
        }
    }
    return true;
}
```

Эта простая функция проверяет, все ли элементы в векторе целых чисел имеют заданное значение. Довольно полезная функция. Настолько полезная, что ее более общая версия уже имеется в заголовке `<algorithm>`:

```
template< class InputIt, class UnaryPredicate >
constexpr bool all_of( InputIt first, InputIt last,
                      UnaryPredicate p );
```

Или, если вы хотите выглядеть головокружительно современным и использовать диапазоны:

```
template< ranges::input_range R, class Proj = std::identity,
          std::indirect_unary_predicate<
              std::projected<ranges::iterator_t<R>,Proj>> Pred >
constexpr bool all_of( R&& r, Pred pred, Proj proj = {} );
```

Имя `all_of` лучше знакомо сообществу программистов, чем `all_matching`. К тому же оно более абстрактное. Не изобретайте это конкретное колесо и следите за простыми циклами. Я не первый, кто это говорит. В 2013 году Шон Пэрент (Sean Parent) подробно муссировал эту тему, выступая с легендарным докладом под названием *Code Seasoning*¹.

В частности, обратите внимание на рекомендацию «Т.2. Используйте шаблоны для выражения алгоритмов, применимых к аргументам многих типов». Стандартные алгоритмы являются безупречным примером такого подхода.

¹ <https://learn.microsoft.com/en-us/events/goingnative-2013/cpp-seasoning>

ШАБЛОНЫ КЛАССОВ И АБСТРАКЦИЯ

В дополнение к шаблонам функций стандартная библиотека содержит несколько полезных шаблонов классов. Их гораздо меньше, чем шаблонов функций. Наиболее широко используемыми, пожалуй, являются `std::vector`, `std::string` и `std::unordered_map`. В процессе рассмотрения находится еще несколько предложений по добавлению дополнительных контейнеров, но они используются гораздо реже, чем шаблоны функций.

В любом курсе обучения программированию предлагаются темы, посвященные структурам данных и алгоритмам. Это основа программирования. Зная перечень доступных структур данных и алгоритмов, вы сможете правильно принимать решения: когда определять новые, адаптировать или просто использовать существующие структуры. Это великая сила, доступная только тем, кто действительно знает, что к чему, правильно называя вещи своими именами и в прямом, и в переносном смысле.

Даже беглого взгляда на стандартную библиотеку достаточно, чтобы заметить, что большую ее часть составляют шаблоны классов и функций. Это объясняется тем, что шаблоны позволяют давать имена сущностям, не беспокоясь о типах. Вам может не нравиться название «вектор» (мне так уж точно), но оно не содержит информации о том, какие данные может хранить эта структура. Для контейнера это неважно. Он просто хранит данные определенным образом, гарантируя, что они будут расположены в памяти в виде непрерывного блока.

Следование соглашениям стандартной библиотеки о контейнерах позволяет создавать структуры данных, подходящие для самых разных решений и хорошо совместимые с существующими алгоритмами. Определите итераторы для своего контейнера, чтобы можно было делегировать поиск и сортировку стандартной библиотеке, или предложите специализации, оптимизированные для вашей структуры данных. Определите соответствующие типы элементов так же, как это делают `std::vector` и `std::unordered_map`. Реализуйте специализацию `std::swap` и, для совместимости с C++20, трехсторонний оператор сравнения. Это упростит изучение и повторное использование ваших контейнеров и позволит вашим клиентам думать о том, как хранить данные, а не о том, как правильно работать с контейнером или какие типы они могут иметь.

Эта независимость от типов позволяет описывать в коде то, что он делает, а не то, с чем он работает. Эту поддержку также предлагает ключевое слово

`auto`. Оно устраниет из кода ненужные детали реализации. Фактически полное определение `std::string` выглядит так:

```
namespace std {
    template<
        class CharT,
        class Traits = std::char_traits<CharT>,
        class Allocator = std::allocator<CharT>
    > class basic_string;
    using string = basic_string<char>;
}
```

Это довольно длинное определение, чтобы держать его в голове, когда основной интерес в действительности представляет только `std::string`. Вам едва ли когда-нибудь придется использовать тип `std::char_traits` или как-то беспокоиться о нем. Точно так же большинству из вас никогда не придется заниматься разработкой механизмов распределения памяти (аллокаторов — `allocator`), разве что вдруг доведется писать код, действующий ближе к голому железу, поэтому типа `std::allocator` обычно вполне достаточно.

Я радуюсь, когда нахожу абстракцию, так как это означает, что я выделил часть предметной области и теперь могу дать ей название. А если мне особенно повезет, то это имя появится во множестве мест в коде.

Широкое использование `std::string` также решает одну из проблем, перечисленных в Core Guidelines и связанных с использованием шаблонов для повышения уровня абстракции кода, — проблему повторного использования. Я радуюсь, когда нахожу абстракцию, так как это означает, что я выделил часть предметной области и теперь могу дать ей название. А если мне особенно повезет, то это имя появится во множестве мест в коде.

`std::string` — отличное имя. Есть много других вещей, которым можно было бы дать названия. Например, комитет по стандартизации мог бы не утвердить объявления `using`, и тогда нам пришлось бы писать определения `std::basic_string<char>`, а затем применять собственные объявления `using`, создавая нагромождения идентичных типов, таких как `char_string`, `ch_string`, `string_char`, `string_c` и т. д. Стока — это распространенный тип данных, поддерживаемый многими языками программирования для обозначения набора символов, представляющего текст. Существуют даже каламбуры на доступные имена. Веревкой (*rope*) или шнуром (*cord*) иногда

называют структуру данных, состоящую из коротких строк и используемую для управления очень длинными строками, такими как статьи или книги. Если вам доводилось участвовать в разработке текстового редактора, то, вероятно, вы использовали такие структуры данных для быстрой вставки и удаления.

`std::string` — это выдающийся пример успеха именования в стандарте. К сожалению, в нем есть и примеры неудач.

ВЫБОР ИМЕНИ – СЛОЖНАЯ ЗАДАЧА

Давайте поговорим о причинах, почему мне не нравится название «вектор», и двухэтапном процессе повышения уровня абстракции с помощью шаблона.

Я получил математическое образование в Сассекском университете. Моей самой сильной областью была абстрактная алгебра. Я помню учебную программу первого семестра, включающую исчисление, анализ, линейную алгебру и введение в философию (это был немного необычный для своего времени британский университет). Линейную алгебру вел профессор Мартин Данвуди (Martin Dunwoody), и каждая его лекция доставляла высшее удовольствие наглядными объяснениями того, как все работает.

Фундаментальным объектом линейной алгебры является вектор. Это n -кортеж значений с сопутствующим набором арифметических операций: сложение, умножение на скаляр и умножение на вектор, известное также как внутреннее произведение. Эти арифметические операции позволяют выполнять самые разные действия: решать системы уравнений, моделировать декартову геометрию. Фактически большая часть современной математики берет начало в линейной алгебре.

Тип `std::vector` — это нечто другое: динамический массив чего угодно, но не фиксированный массив числовых значений. Для него не определены векторные операции, поэтому вы не сможете складывать или умножать их. На самом деле единственное сходство `std::vector` с вектором заключается в хранении нескольких элементов. Такой выбор не совсем правильного названия для одного из наиболее широко используемых контейнеров ставит язык, имеющий в своем арсенале это имя, в странное положение.

В повседневной работе я моделирую вымышленные миры и использую линейную алгебру для имитации их физической реальности. Честно говоря, я легко могу определить свой тип вектора. Например, в моей библиотеке имеется `gdg::vector_3f` — вектор, хранящий три числа с плавающей точкой. Он используется для определения местоположения некоторой точки в трехмерном пространстве, а также для представления скорости и ускорения. Однако имя `std::vector` в коде означает совсем не то, о чем мог бы подумать любой математик. Мы настолько привыкли к этому имени, что не замечаем его некоторой неправильности, но оно может вызвать затруднение у тех, кто приходит в C++ из математики или из других языков. Возможно, этому типу лучше подошло бы имя `dynarray`. Формирование абстракции — это двухэтапный процесс. На первом этапе идентифицируется абстракция, исследуются ее API, входные и выходные данные, связь с предметной областью. А на втором этапе ей дается имя. Поиските литературу, посвященную этой теме, помните об ответственности, которую вы несете, и о последствиях, которые могут наступить в результате выбора вами имени для какой-то сущности.

ПОДВЕДЕМ ИТОГ

- Шаблоны функций и классов скрывают детали и, следовательно, сложность за осмысленными именами.
- Многие шаблоны функций в стандартной библиотеке могут заменять циклы.
- Многие алгоритмы в стандартной библиотеке могут использовать существующие структуры данных.
- Выражение абстракций в виде шаблонов классов обеспечивает более широкое повторное использование кода, особенно при хорошем выборе имен.

ГЛАВА 4.6

Т.10. Задавайте концепции для всех аргументов шаблона

КАК МЫ ЗДЕСЬ ОКАЗАЛИСЬ?

Концепции (или концепты – concepts) — одна из важнейших составляющих C++20. Им потребовалось очень много времени, чтобы пробиться в стандарт. Первая попытка стандартизации концепций была предпринята сразу после публикации C++98, она почти увенчалась успехом в C++11, или C++0x, как его называли в то время. Однако в самый последний момент концепции были изъяты из рабочего проекта и перенесены в Технические спецификации (Technical Specification, TS). Технические спецификации используются для сбора опыта внедрения. Технические спецификации концепций были опубликованы в конце 2015 года, и это позволило разработчикам компиляторов проверить на деле техническую пригодность этого инструмента.

В 2016 году прием новых предложений для добавления в C++17 был остановлен, но к этому моменту концепции были недостаточно проверены для добавления в официальный язык. Однако на первом заседании комитета в Торонто летом 2017 года, посвященном началу цикла подготовки C++20, предложение 11 «О применении изменения P0734 (Wording Paper, C++ extensions for Concepts) к рабочему документу C++» прошло под дружные аплодисменты.

Кстати, это было первое заседание комитета с моим участием¹. Я и раньше посещал собрания исследовательских групп и национальных органов, но впервые оказался в одной комнате с людьми, определяющими стандарты, и не мог обойти этот факт вниманием, не рассказав о нем.

¹ Рассказ Дж. Гая Дэвидсона.

Продвижение тогда получили не только концепции. На голосованиях национальных органов прошло также предложение 12 о введении модулей, редакционный комитет одобрил предложение 13 о включении сопрограмм в технические спецификации, а в предложении 22 было поручено редакционному комитету, включая меня, проверить правильность технической спецификации диапазонов. Было ясно, что C++20 должен стать большим рывком вперед. Я не ожидал такого наплыва событий от своего первого участия в заседании комитета, но историю делают не «лежачие камни», дорогу осилит идущий. Если вы, читатель, когда-нибудь окажетесь на своем первом собрании комитета, найдите его и предложите свои инициативы касательно развития языка.

Итак, вернемся к концепциям и рассмотрим проблему, которую они предназначены решить, на примере следующего шаблона функции:

```
template <typename T>
T const& lesser(T const& t1, T const& t2) {
    return t1 < t2 ? t1 : t2;
}
```

Обладай программист минимальным везением, и судьба избавила бы его от необходимости писать такой код, ведь всегда можно обратиться к `std::min` из заголовка `<algorithm>`. Кстати, мы использовали имя `lesser`, так как `min` чрезвычайно перегружено в каждой реализации.

Хитрость заключается в том, что эту функцию можно вызвать только с определенными типами: если ей передать пару ссылок на `std::unordered_map<string, string>`, то компилятор недвусмысленно сообщит, что из этого ничего не получится. С этими типами операндов нет совпадения для оператора `operator<`. Сообщение об ошибке недостаточно подробное, но все же позволяет понять суть. Вот отредактированный образец листинга ошибок, созданный компилятором GCC:

```
<source>: In instantiation of 'const T& lesser(const T&, const T&)
           [with T = std::unordered_map<int, int>]':
<source>:14:31:   required from here
<source>:6:15: error: no match for 'operator<' (operand types are
           'const std::unordered_map<int, int>' and 'const std::unordered_map<int,
           int>')
           6 |     return t1 < t2 ?t1 : t2;
             |     ~~~^~~~
In file included from ...:
<filename>:1149:5: 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>&)'
1149 |     operator<(const __normal_iterator<_IteratorL, _Container>& __lhs,
|     ^~~~~~

```

В сообщении об ошибках говорится, что при попытке создания экземпляра шаблона функции `lesser` с использованием `std::unordered_map<int, int>` в качестве параметра шаблона компилятор не нашел подходящего оператора `operator<`. Затем компилятор перечислил и все операторы-кандидаты (мы в свое время остановились на одном, но когда попробовали его в Compiler Explorer, то компилятор перечислил довольно много кандидатов), чтобы показать отсутствие требуемого.

Эта простая ошибка в функции из четырех строк породила 122-строчное сообщение. В более реалистичных функциях объем сообщения об ошибке значительно увеличивается, и становится все труднее определить основную причину проблемы. Если первую попытку создать экземпляр и ошибку разделяет длинный стек вызовов шаблонов функций, то вы можете потратить довольно много времени на изучение выводимой информации.

Эта проблема была всегда, но есть способы уменьшить ее. Конечно, мы не можем определить перегруженные версии функции для всех мыслимых типов, но можем перегрузить набор типов, используя `std::enable_if`. Чтобы пойти на такой шаг, вы должны по-настоящему ненавидеть себя и своих коллег. Вот как это выглядит:

```

template <typename T,
          std::enable_if_t<std::is_arithmetic<T>::value, bool> = true>
T const& lesser(T const& t1, T const& t2);

```

Имя `std::is_arithmetic` определено в заголовке `<type_traits>` вместе с набором других типажей (`traits`)¹, таких как `std::is_const` и `std::is_trivially_constructible`. Столь перегруженная версия будет выбрана для любого арифметического типа (целочисленного или с плавающей точкой). Сам код не особенно понятен, если только вы не сталкивались с этим приемом раньше. Такое программное решение считается программированием «экспертного уровня», и обычно опытные преподаватели языка откладывают рассмотрение таких примеров до тех пор, пока их ученики полностью не освоют параметры шаблонов по умолчанию и принцип

¹ Иногда их называют трейтами или свойствами. — Примеч. пер.

«Ошибка подстановки не является ошибкой», или SFINAE (Substitution Failure Is Not An Error).

Мы не будем подробно останавливаться на толковании этого приема, потому что теперь можно поступить проще.

ОГРАНИЧЕНИЕ ПАРАМЕТРОВ

Проблема с идиомой `std::enable_if` в том, что это слишком многословный способ ограничения функции на уровне библиотеки. Ограничение замаскировано в мешанине знаков препинания. Важной частью ограничения является имя `std::is_arithmetic`. Остальное — просто подпорки, помогающие преодолеть недостатки языка.

В действительности нужна возможность выразить что-то вроде этого:

```
template <typename std::is_arithmetic T>
T const& lesser(T const& t1, T const& t2);
```

Ограничение определяется рядом с типом. Это более понятный синтаксис. К сожалению, он не поддерживается в C++. Он слишком неоднозначный для компилятора и рискует превратиться в такую же мешанину, если потребуется применить несколько ограничений. Попробуем другую версию:

```
template <typename T>
where std::is_arithmetic<T>::value
T const& lesser(T const& t1, T const& t2);
```

Теперь ограничение стало определяться в отдельной строке между объявлением шаблона и именем функции. Но этот синтаксис тоже не поддерживается в C++: ключевое слово `where` слишком напоминает SQL, поэтому мы заменяем его на `requires`:

```
template <typename T>
requires std::is_arithmetic_v<T>
T const& lesser(T const& t1, T const& t2);
```

Здесь синтаксис стал допустимым для C++20. Если потребуется добавить дополнительные ограничения, их можно включить в предложение `requires`, например, так:

```
template <typename T>
requires std::is_trivially_constructible_v<T>
and std::is_nothrow_move_constructible_v<T>
T f1(T t1, T t2);
```

Можно удовлетвориться использованием слова `and` в этом примере. Но, коль скоро это логическая конъюнкция, не лучше ли использовать `&&`?

Это уже дело вкуса и стиля. Вы можете использовать любой из вариантов. Если вам больше нравятся знаки и символы, а не `and`, используйте их. Однако выражения на простом английском языке кажутся людям более понятными.

Но ведь и этот синтаксис выглядит довольно громоздким. Ключевое слово `template` необходимо, чтобы подчеркнуть, что данное объявление является шаблоном функции, но наличие ограничений также подразумевает, что и объявление является шаблоном. Так ли уж нужны они оба? Разве не проще выглядел бы такой синтаксис:

```
T f2(std::is_arithmetic<T> t1, std::is_arithmetic<T> t2);
```

Он выглядит как объявление функции, принимающей значения любых арифметических типов. К сожалению, в этом случае снова возникает проблема с неоднозначностью. Первый экземпляр `T` выглядит как определение, а не как возвращаемый тип. Оно появляется из ниоткуда, но мы можем заменить его на `auto`, так как это шаблон функции, и затем с уверенностью предположить, что это определение, видимо, и подразумевает возвращаемый тип. Последующее использование `T` тоже создает проблему. Как показать, что передается ограниченный тип?

В C++14 появилось понятие обобщенных лямбда-выражений. Вместо явного определения типа для каждого параметра можно указать `auto` и позволить компилятору самому определить тип. Здесь мы можем использовать ту же форму:

```
auto f3(std::equality_comparable auto t1, std::equality_comparable
auto t2);
```

Появилось одно малозаметное отличие: больше не требуется, чтобы `t1` и `t2` были одного типа или чтобы любой тип соответствовал возвращаемому типу, как в случае с `f1`. Единственное ограничение: оба должны удовлетворять условию `std::equality_comparable`. Если потребуется, чтобы оба параметра были одного типа, можно объявить тип `t2` как `decltype(t1)`:

```
auto f3(std::equality_comparable auto t1, decltype(t1) t2);
```

Кажется, немного неудобно; однако для функций, принимающих только один параметр, использование `auto` становится гораздо более привлекательным вариантом.

Обратите также внимание, что теперь вместо `std::is_arithmetic` передается `std::equality_comparable`. Почему? Потому что `std::is_arithmetic` — это шаблон класса, а `std::equality_comparable` — концепция. Такой синтаксис поддерживается только для концепций. Вскоре мы увидим, как его можно грамотно использовать.

Этот вариант сообщает пользователю больше информации в гораздо меньшем объеме кода. Прочитав сигнатуру функции, программист сразу увидит, что это шаблон функции, ограниченный типами, поддерживающими сравнение на равенство. Кроме того, если он попытается вызвать функцию, используя аргументы, не поддерживающие сравнение на равенство, то компилятор сразу сделает заключение, что предоставленные аргументы не являются сравнимыми на равенство, как того требует объявление.

К сожалению, если вам потребуется добавить дополнительные ограничения к параметрам, такой синтаксис вас не устроит. Он станет выглядеть так:

```
auto f4(std::copyable and std::default_initializable auto t);
```

Наличие `auto` означает, что задается ограничение. Но первое ограничение неоднозначно. Чтобы избавиться от этого, можно определить свою концепцию:

```
template <typename T>
concept cdi = std::copyable <T>
and std::default_initializable <T>;
auto f4(cdi auto t);
```

Конечно, `cdi` — ужасное имя. Возможно, мы уже упоминали об этом, но выбор говорящих имен — сложная задача. Концепции особенно щекотливы в этом отношении. Имя типа `is_copyable_and_default_initializable` — плохое имя, потому что просто перемещает проблему в другое место. Что на самом деле оно означает?

В данном случае мы прекрасно знаем, что под этим именем подразумевается полурегулярный тип, который можно копировать и инициализировать значением по умолчанию. Это хорошо известная часть таксономии типов, и стандартная библиотека предоставляет эту концепцию наряду с набором других в заголовке `<concepts>`. Узнать чуть больше о таксономии типов можно в блоге Райнера Гrimма (Rainer Grimm)¹.

¹ <https://www.modernescpp.com/index.php/c-20-define-the-concept-regular-and-semiregular>

Заголовок `<concepts>` заслуживает особого внимания, потому что показывает, как концепции основываются друг на друге. Концепция `std::semiregular` состоит из концепций `std::copyable` и `std::default_initializable`. Концепция `std::copyable` состоит из `std::copy_constructible`, `std::movable` и `std::assignable_from`.

Важно вникнуть в таксономию типов, чтобы понять, например, разницу между регулярными и полурегулярными типами. Эти термины ввел Алекс Степанов (Alex Stepanov), создатель стандартной библиотеки шаблонов (Standard Template Library, STL), он обсуждает их в своей книге *Elements of Programming*¹.

КАК АБСТРАГИРОВАТЬ СВОИ КОНЦЕПЦИИ

Здесь мы имеем дело уже с проблемой абстракции. Имя `is_copyable_and_default_initializable` просто повторяет вопрос: «Как назвать эту концепцию?» Это не абстракция. Повышение уровня абстракции означает обход таксономии типов и выбор говорящего имени для того, что образуется на пересечении этих концепций.

Это основа абстрагирования. Выше, в главе 2.1 «Р.11. Инкапсулируйте беспорядочные конструкции, а не разбрасывайте их по всему коду», мы описали разницу между инкапсуляцией, скрытием данных и абстракцией. Давать концепции имя по частям, из которых она состоит, — это инкапсуляция, а не абстрагирование. К тому же такой способ именования не масштабируется. Концепции должны составляться так же, как они составляются в заголовке `<concepts>`. Этот механизм композиции концепций покажет уровень абстракции, на котором вы работаете, и отразит разложение на составляющие вашей области решений.

Определяя шаблоны функций, вы пытаетесь инкапсулировать общий алгоритм. Концепции, подходящие для вашей функции, абстрагируются от этого алгоритма. Подобное может означать, что имена некоторых концепций могут просто отражать имя функции. Давайте разработаем тривиальную функцию сортировки:

```
template <typename InIt>
void sort(InIt begin, InIt end)
{
```

¹ Stepanov A., McJones P. Elements of Programming. — Boston: Addison-Wesley, 2009
(Степанов А., Мак-Джоунс П. Начала программирования).

```

while (begin != end)           // ①
{
    auto src = begin;          // ②
    auto next = std::next(begin); // ③
    while (next != end)
    {
        if (*next < *src) {      // ④
            std::iter_swap(src++, next++);
        }
    }
    --end;
}
}

```

Эта функция реализует алгоритм пузырьковой сортировки, временная сложность которого, как известно, является квадратичной, поэтому он почти не используется на практике, но вполне соответствует нашим целям. Что мы знаем о необходимых свойствах параметра шаблона?

В ① выполняется проверка на равенство двух экземпляров `InIt`. Это означает, что они должны удовлетворять условию равенства. В ② производится копирование, а значит, они должны поддерживать копирование. В ③ вызывается `std::next`, следовательно, они должны поддерживать операцию инкремента. В ④ производится разыменование и сравнение итерируемого типа. Давайте посмотрим, что нам доступно в стандарте, и попробуем подобрать имя для этого набора требований к типам.

Как оказывается, у нас богатый выбор. Стандартная библиотека усыпана концепциями, и есть даже целый заголовочный файл, содержащий некоторые фундаментальные концепции. Например, в ① нам нужна возможность сравнивать итераторы. Это означает, что нам нужна концепция `std::equality_comparable` из заголовка `<concepts>`. Операция присваивания в ② поддерживается концепцией `std::copy_constructible`, а ③ поддерживается концепцией `std::incrementable` из заголовка `<iterator>`.

Но самое интересное начинается в ④. Здесь происходит разыменование итераторов и сравнение значений, на которые они указывают, после чего производится разыменование итераторов и обмен значениями. Это неявное поведение поддерживается `std::indirectly_readable` из `<iterator>` и `std::swappable` из `<concepts>`. Сравнение значений производится немного сложнее: нужно вызвать предикат сравнения, определяющий отношение двух operandов. И снова на помощь приходит `<concepts>` с концепцией `std::invocable`, которая требуется концепции `std::predicate`, которая

требуется концепции `std::relation`, а она, в свою очередь, требуется концепции `std::strict_weak_order`.

Все это в совокупности дает нам отличную отправную точку для нашей концепции параметров этого шаблона функции.

```
template <typename R, typename T, typename U>
concept sort_concept = std::equality_comparable<T, U>
and std::copy_constructible<T>
and std::incrementable<T>
and std::indirectly_readable<T>
and std::swappable<T>
and std::strict_weak_order<R, T, U>;

void sort(sort_concept auto begin, sort_concept auto end);
```

Надеемся, вы согласитесь, что `sort_concept` — ужасное имя.

А как насчет `sortable`?

После замены сигнатура функции приобретает следующий вид:

```
void sort(sortable auto begin, sortable auto end);
```

Мы избавились от угловых скобок, что стало явным облегчением. Нам известно, что это шаблон функции, из списка параметров имя-`auto`-имя. Если передать что-то, что не удовлетворяет концепции `sortable`, то компилятор сможет, например, заявить, что «этот тип не поддерживает концепцию `sortable`: не удовлетворяет `std::swappable`».

Вас не должно удивлять, что `std::sortable` уже является концепцией, определенной в заголовке `<iterator>`. Желающие узнать, как она сконструирована, могут обратиться к документации по `std::sortable` на сайте [cppreference.com](https://en.cppreference.com)¹. Приведенная декомпозиция `sortable` из примитивной функции `sort` была лишь первым шагом, а не полным решением. Вы обнаружите при детальном рассмотрении, что в концепции `sortable` гораздо больше компонентов. Например, мы опустили `std::assignable_from`. Однако рано или поздно кто-то попытается создать экземпляр функции с типом, не удовлетворяющим этой концепции, и компиляция завершится ошибкой. После анализа сообщения об ошибке можно будет улучшить определение `sortable`, добавив уже это новое понятие.

¹ <https://en.cppreference.com/w/cpp/iterator/sortable>

Указав концепцию аргументов шаблона, вы уменьшите количество букв и знаков пунктуации и кристаллизуете смысл в наиболее подходящей для этого точке. Она подчеркнет уровень абстракции, на котором вы работаете. Она сделает ваш API более понятным, выразительным и удобным в использовании за счет лучшей документации, лучших сообщений об ошибках и более точной перегрузки¹.

РАЗЛОЖЕНИЕ НА СОСТАВЛЯЮЩИЕ ЧЕРЕЗ КОНЦЕПЦИИ

Обратите внимание, что имя концепции в примере выше было выбрано на основе алгоритма, а не типа. Концепции находятся в алгоритмах, а не в типах. Это дополнительно подчеркивается ответом на вопрос: «Как создавать концепции, характерные для предметной области?»

Например, если у вас есть масса удобных шаблонов функций для выполнения операций при вычислении заработной платы, то все они могут потребовать, чтобы экземпляр параметра типа шаблона поддерживал выполнение соответствующих запросов расчета заработной платы. Для определения таких фундаментальных концепций используется ключевое слово `requires`:

```
template <typename T>
concept payroll_queryable =
    requires (T p) { p.query_payroll(); };

void current_salary(payroll_queryable auto employee);
```

Понятие абстрагируется от того, что можно сделать, а не от свойства типа. Оно описывает поведение, а не свойство. Однако осторегайтесь чрезмерного ограничения. Представьте, что функция реализована так:

```
void current_salary(payroll_queryable auto employee)
{
    std::cout << "unimplemented\n";
}
```

Такой вызов функции сгенерирует ошибку компиляции:

```
current_salary("Beth");
```

¹ См. *Строуструп Бьорн*. Процветание в переполненном и меняющемся мире: C++ 2006–2020. ACM/SIGPLAN Конференция по истории языков программирования, HOPL-IV. Лондон. Июнь 2020 года. <https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf>.

Даже притом, что реализация шаблонной функции не использует возможности, предлагаемые этой концепцией, компилятор все равно сообщит, что ограничения не соблюдены.

Когда при реализации решения обнаруживаются абстракции, описывайте их не только с точки зрения типов и функций, но также с точки зрения типов и поведения. Прелест концепций в том, что они позволяют построить таксономию для предметной области решения. Стандарт не ограничивает вас определенными типами, и точно так же он не ограничивает вас определенными концепциями. Вы можете определять свои собственные концепции так же, как и типы, и использовать их для уточнения особенностей вашей библиотеки, отражая ее назначение и полезность. Процесс разложения области решения на управляемые части и поиска составляющих значительно упрощается наличием механизма, с помощью которого можно идентифицировать поведение и типы.

Единственное, что сложнее выбора имен, — это таксономия, то есть выбор названий имен. Концепции позволяют определять идентификаторы поведения для вашей области решения. Конечно, не нужно сходить с ума и идентифицировать все подряд, это совершенно не обязательно. Концепции ограничивают параметры шаблонов, поэтому если ваша программа не состоит целиком из шаблонов функций, то вы не сможете использовать их все. Это напоминает аргумент, которым пользуются противники объектно-ориентированного программирования, заявляя, что эта парадигма требует моделировать каждую деталь. Однако для полной ясности укажите ограничения в виде концепций для всех аргументов вашего шаблона.

ПОДВЕДЕМ ИТОГ

Всякий раз, начиная читать лекции о C++, преподаватели заявляют, что есть две вещи, о которых нужно беспокоиться: состояние и выполнение. За синтаксическим шумом объявлений классов и уровней доступа, специализаций шаблонов функций и квалификации констант файлы с исходным кодом на C++ определяют объекты и функции. Компоновщики связывают объекты и функции в один двоичный файл. Операционные системы связывают вместе файлы и процессы. Курсы информатики учат структурам данных и алгоритмам. Теперь у нас есть еще одна пара для обсуждения: типы и концепции.

Определяя шаблоны функций, проанализируйте все действия, выполняемые с параметрами, и абстрагируйте от них свои ограничения. Определите требования и повторно используйте готовые или сконструируйте новые концепции и опишите их в точке, где принимается аргумент.

Концепции находятся в алгоритмах, а не в типах.

Используйте концепции, чтобы сообщать пользователям, какие типы можно передать, и помогать компилятору генерировать более понятные диагностические сообщения.

По возможности используйте стандартные концепции, они совместимы и вызывают меньше разногласий. Но при необходимости определяйте и свои концепции, характерные для предметной области, в которой вы работаете.

В Core Guidelines есть несколько рекомендаций, которые касаются концепций, в том числе:

- «Т.10. Определяйте концепции для всех параметров шаблона»;
- «Т.11. По возможности используйте стандартные концепции»;
- «Т.26. Определяйте концепции с точки зрения сценариев использования, а не простого синтаксиса»;
- «Т.121. Используйте метапрограммирование шаблонов в первую очередь для имитации концепций».

Концепции являются одной из важнейших особенностей C++20, и процесс их разработки уже дал некоторое представление об их правильном использовании. По мере их распространения в сообществе C++ будут появляться новые статьи в блогах и рекомендации в Core Guidelines. Следите за обновлениями в них.

V

ПИШИТЕ ХОРОШИЙ КОД ПО УМОЛЧАНИЮ

Глава 5.1 Р.4. В идеале программа должна быть статически типобезопасной.

Глава 5.2 Р.10. Неизменяемые данные предпочтительнее изменяемых.

Глава 5.3 I.30. Инкапсулируйте нарушения правил.

Глава 5.4 ES.22. Не объявляйте переменные, пока не получите значения для их инициализации.

Глава 5.5 Рег.7. При проектировании учитывайте возможность последующей оптимизации.

Глава 5.6 E.6. Используйте идиому RAII для предотвращения утечек памяти.

ГЛАВА 5.1

Р.4. В идеале программа должна быть статически типобезопасной

БЕЗОПАСНОСТЬ ТИПОВ – ЭТО СРЕДСТВО ЗАЩИТЫ В C++

Безопасность типов считается такой же важной особенностью C++, как и детерминированность уничтожения объектов. Безопасность типов сообщает различным частям программы, что означает последовательность битов. Например, вот последовательность битов для числа 1729 в виде 32-битного целого со знаком:

```
std::int32_t Ramanujan_i = 0b0000'0000'0000'0000'0000'0110'1100'0001;
```

А вот последовательность битов для числа 1729 в виде 32-битного целого без знака:

```
std::uint32_t Ramanujan_u = 0b0000'0000'0000'0000'0000'0110'1100'0001;
```

Как видите, они идентичны. Но взгляните на последовательность битов, представляющую число 1729 в виде значений с плавающей точкой:

```
float Ramanujan_f = 0b0100'0100'1101'1000'0010'0000'0000'0000;
```

Если эту последовательность битов интерпретировать как 32-битное целое со знаком, то получится значение 1 155 014 656. И наоборот, если Ramanujan_i интерпретировать как значение с плавающей точкой, то получится $2,423 \times 10^{-42}$.

Другие языки используют так называемую «утиную типизацию». Они исходят из предпосылки, что программист всегда точно знает, что он делает.

Таким образом, если программист потребует умножить строку на цвет, языки должны предоставить ему такую возможность. Интерпретатор приложит все усилия, чтобы удовлетворить это требование, и выдаст ошибку во время выполнения, если у него это не получится. Но ошибки времени компиляции предпочтительнее ошибок времени выполнения, потому что для их выявления не требуется тестировать код. Поддержка безопасности типов устраниет целый класс ошибок.

Стандарт C++ старается максимально помочь писать типобезопасный и в то же время читабельный код. Если вы вызываете функцию, которая принимает параметр `long`, но передаете ей аргумент `short`, то компилятор знает, как сгенерировать код, чтобы исправить это несоответствие, не создавая ошибок. Если вы передадите аргумент с плавающей точкой, то компилятор молча вставит вызов функции, округляющей число до ближайшего целого, и преобразует полученную последовательность битов в соответствующее целочисленное представление. Однако это довольно затратная функция. Случалось, программист шерстил вывод, сгенерированный компилятором для функции `_ftol`, чтобы убедиться, что она никогда не была вызвана неявно.

Более того, перегрузка позволяет создавать версии функций для разных типов, а шаблоны функций — определять алгоритмы, не зависящие от типов. Как мы видели в главе 1.2, предпочтение, отданное использованию перегруженной версии, требует учитывать множество нюансов, вплоть до выбора некоторых преобразований между типами, чтобы обеспечить передачу правильных аргументов в параметрах функции. Функция `_ftol` является примером такого поведения.

Ключевое слово `auto` устраняет проблемы с типом, позволяя определять типы по коду, который вы пишете. Если вы напишете:

```
auto Ramanujan = 1729;
```

то компилятор будет интерпретировать `Ramanujan` как целое число и заполнит память, принадлежащую определению, правильной последовательностью битов. Точно так же, если вы напишете:

```
auto Ramanujan = 1729u;
```

или

```
auto Ramanujan = 1729.f;
```

компилятор определит тип по операции присваивания (`unsigned int` и `float` соответственно) и правильно инициализирует память.

Вся эта машинерия существует для того, чтобы избавить вас от необходимости заботиться о внутреннем представлении. Чем меньше времени вы тратите на программирование с учетом типов и чем больше — на программирование интерфейсов, тем безопаснее ваш код.

Поддержка безопасности типов устраниет целый класс ошибок.

ОБЪЕДИНЕНИЯ

Подорвать безопасность типов можно несколькими способами, большинство из которых унаследовано от C. Рассмотрим ключевое слово `union`.

```
union converter {
    float f;
    std::uint32_t ui;
};
```

Здесь определяется тип, содержимое которого может интерпретироваться как `float` или как `unsigned integer`. Это путь к огорчениям и погибели. Пользуясь таким определением, программист должен всегда знать и отслеживать, какой тип имеет представляемая переменная. Он имеет полное право написать, например, такой код:

```
void f1(int);
void f2(float);

void bad_things_happen() {
    converter c;
    c.ui = 1729;
    f1(c.ui);
    f2(c.f);
}
```

В вызов `f2` компилятор передаст значение $2,423 \times 10^{-42}$, которое *может* совпадать с требуемым, но это маловероятно.

Правильный способ организовать хранение значений разных типов в одном объекте — использовать тип `std::variant` из стандартной библиотеки C++17. Это так называемое размеченное объединение, то есть вариантный объект знает, значение какого типа он хранит. Все в полном соответствии

с рекомендацией из Core Guideline: «C.181. Избегайте простых объединений». Вот как можно его использовать:

```
#include <variant>

std::variant<int, float> v;
v = 12;
float f = std::get<float>(v); // вызовет исключение std::bad_variant_access
```

Вы можете посчитать это иллюстративным примером, но, хотите — верьте, хотите — нет, такое объединение уже использовалось много лет назад, чтобы быстро вычислить обратный квадратный корень:

```
float reciprocal_square_root(float val) {
    union converter {
        float f;
        std::uint32_t i;
    };

    converter c = {.f = val};
    c.i = 0x5f3759df - (c.i >> 1);
    c.f *= 1.5f - (0.5f * val * c.f * c.f);
    return c.f;
}
```

Чтобы получить высокую скорость умножения за счет сдвига битов, этот недостойный подражания код использует особенности реализации представлений типа `float`, которые считаются неопределенным поведением в C++. Набор инструкций SSE, к счастью, сделал это избыточным благодаря добавлению инструкции `rsqrts`, но, начиная с C++20, то же самое можно сделать правильно, используя `std::bit_cast`:

```
float Q_rsqrt(float val)
{
    auto half_val = val * 0.5f;
    auto i = std::bit_cast<std::uint32_t>(val);
    i = 0x5f3759df - (i >> 1);
    val = std::bit_cast<float>(i);
    val *= 1.5f - (half_val * val * val);
    return val;
}
```

Такая возможность приведения по-прежнему противоречит типобезопасному программированию и позволяет нам плавно перейти к следующему разделу.

ПРИВЕДЕНИЕ

Приведение — операция изменения типа объекта. Этот общий термин в информатике имеет более точный набор смысловых значений в C++. Изменение типа объекта может повредить статической безопасности типов. А иногда это совершенно безопасно, как в следующем примере:

```
short f2();
long result = f2();
```

Здесь результат вызова `f2` преобразуется из типа `short` в тип `long` с помощью неявного преобразования. Это целочисленное расширение, которое просто расширяет представление. Каждое значение, которое может принимать объект типа `short`, также может быть представлено объектом типа `long`. То же верно и при расширении значения типа `float` до типа `double`.

Но не все преобразования настолько безопасны. Например, такой случай:

```
long f2();
short result = (short)f2();
```

Это явное преобразование, в котором указан целевой тип. Если известно, что возвращаемое значение функции `f2` находится в пределах диапазона представления объекта `short`, то этот код условно безопасный. Но он не совсем безопасный: спецификация функции может измениться и она начнет возвращать большие числа. От компиляторов не требуется выдавать диагностические сообщения о выполнении рискованного преобразования, хотя большинство из них все-таки выдадут предупреждение.

Такой стиль преобразования часто называют приведением в стиле C. Этот синтаксис берет свое начало в языке программирования C. Преобразования были важны для языка программирования C. Рассмотрим следующую сигнатуру функции:

```
long max(long, long);
```

Если передать этой функции объекты типов `long` и `short` в отсутствие доступного преобразования, то компилятор сгенерирует ошибку и нам придется написать другую функцию:

```
long max_ls(long, short);
```

(Поскольку язык C не поддерживает перегрузку функций, то для определения версии функции, принимающей параметры других типов, вам придется использовать другое имя.) К счастью, мы избавлены от этого

излишнего многословия. А вот в современном C++ еще меньше причин для беспокойства. Благодаря поддержке перегрузки и шаблонов функций можно позволить себе быть более точными в приведении типов.

Приведение действительно может быть довольно рискованным. На самом деле, чтобы подчеркнуть опасность, были введены новые громоздкие ключевые слова:

```
static_cast<T>(expr)      // Выполняет приведение в стиле С
dynamic_cast<T>(expr)    // Приводит к типу выражения expr
const_cast<T>(expr)      // Устраняет квалификатор const
reinterpret_cast<T>(expr) // Повторно интерпретирует
                           // последовательность битов
```

Рекомендация «ES.48. Избегайте приведения» предостерегает от этого. Один вид этих слов должен вселять страх в ваше сердце. Некоторые компиляторы предлагают ключ командной строки, включающий вывод предупреждений о явном приведении типов в стиле C, чтобы вы могли быстро обнаружить потенциальный источник неприятностей. Часто это происходит при попытке вызвать функцию из сторонней библиотеки, написанной не так хорошо, как ваша. Замена всех операций приведения в стиле C вызовами `static_cast` поможет отметить места, где не все в порядке. Ваш компилятор сообщит вам, когда `static_cast` не имеет смысла, например, при приведении типа `int` к `std::pair<std::string, char>`. Так же как и в случае с приведением в стиле C, этим ключевым словом вы заявляете: «Я знаю границы диапазона значений, все будет хорошо».

Далее по списку опасность увеличивается. Ключевое слово `dynamic_cast` позволит выполнить приведение через иерархию наследования к подклассу. Само это описание должно вас насторожить. Набирая на клавиатуре `dynamic_cast`, вы становитесь человеком, который в споре аргументирует так: «Я просто знаю, мне не нужны доказательства». Конечно, `dynamic_cast` может потерпеть неудачу, особенно если уверенность автора не имеет ничего общего с действительностью. В таком случае возвращается нулевой указатель или, если `dynamic_cast` применяется к ссылке, генерируется исключение, и тогда вам придется перехватить это исключение и навести порядок. Это прием с дурным запахом. Вы спрашиваете: «Ты объект типа X?» — но задавать такие вопросы в компетенции системы типов. Явно задавая такой вопрос, вы неявно подрываете ее.

С `const_cast` мы попадаем на территорию «совершенного зла». `const_cast` позволяет отбросить, аннулировать объявление константности или, что случается редко, привести к `const volatile`. Мы исследовали этот тип приведения в главе 3.4, так что вы уже знаете, что использовать его — плохая

идея. Отбрасывая константность, например, для полученной константной ссылки, вы выбиваете почву из-под ног вызывающего. Вы сами объявили в своем API: «Чтобы исключить накладные расходы на копирование, в эту функцию можно без опаски передавать объекты по ссылке, а не по значению, и они останутся неизменными». Это обман: если потребовалось отбросить константность, то такой шаг предполагает необходимость каким-то образом изменить объект, а это противоречит вашему же обещанию. Так вы не найдете верных друзей. Опять же единственный случай, когда допустимо использовать это ключевое слово, — вызов плохо продуманной сторонней библиотеки, не поддерживающей константность. `const_cast` также используется для устранения квалификатора `volatile`; при использовании устаревшего кода с ключевым словом `volatile` применение `const_cast` может быть вполне приемлемым вариантом.

Наконец, с `reinterpret_cast` мы получаем в руки ружье, предназначеннное для того, чтобы выстрелить себе в ногу, и это типичный результат, который дает применение ключевого слова `reinterpret_cast`. Оно выполняет преобразование типов, просто объявляя, что битовая комбинация теперь должна интерпретироваться как нечто другое. Оно не требует процессорного времени (в отличие от `static_cast` или `dynamic_cast`, которые могут вставить функцию времени выполнения, реализующую преобразование) и просто говорит: «Теперь это мое шоу. Мне не нужна безопасность типов в этой части». Некоторые преобразования с `reinterpret_cast` недоступны: с помощью этого ключевого слова нельзя отбросить квалификаторы `const` или `volatile`. Для этого нужно выполнить два последовательных приведения типов:

```
int c = reinterpret_cast<int>(const_cast<MyType&>(f3()));
```

Один вид всего этого синтаксиса должен заставить вас остановиться и поразмыслить. Вы можете вызвать что-то подобное, чтобы создать дескриптор для произвольных типов, но, честно говоря, есть более безопасные способы сделать это.

Мы уже встречались с `std::bit_cast` выше в этой главе. Но самое неприятное, что это не операция на уровне языка, а библиотечный объект. Единственное, что недоступно для `reinterpret_cast`, — приведение типа указателя или ссылки. Для этого нужно использовать `std::bit_cast`. Это последний уровень порочности, настолько далекий от статической безопасности типов, насколько это вообще возможно.

Безопасность типов — это средство защиты. Приведение типов может нарушить безопасность типов. Выделяйте приведение явно там, где оно имеет место, и избегайте его по возможности.

ЦЕЛЫЕ БЕЗ ЗНАКА

Ключевое слово `unsigned` — странный зверь. Оно изменяет целочисленные типы и сигнализирует, что они будут иметь беззнаковое представление, в частности, что тип не использует представление с дополнением до двух. Когда вы прочитаете раздел о приведении типов, это изменение в представлении должно вызывать у вас легкую тошноту.

По нашему опыту программирования можем сказать с достаточной уверенностью, что чаще всего неправильное применение `unsigned` происходит из-за непонимания инвариантов класса. Инженеру может потребоваться представить значение, которое никогда не может быть меньше нуля, и поэтому он выберет беззнаковый тип. Но как утвердить этот инвариант? Следующий прием никогда не подведет:

```
void f1(unsigned int positive)
{
    ... assert(positive >= 0);
}
```

Для чисел меньше нуля нет соответствующего представления, поэтому значения всегда будут больше или равны нулю. Это ключевое слово также сообщает об одной из частых и (наименее) любимых программистами ошибок:

```
for (unsigned int index = my_collection.size(); index >= 0; --index)
{
    ... // программа никогда не покинет этот цикл
}
```

Может показаться хорошей идеей представлять температуру в градусах Кельвина, массу или экранные координаты беззнаковым типом, но, когда понадобится выполнить некоторые арифметические действия, может возникнуть проблема. Вывод этой программы нелогичен:

```
#include <iostream>

int main() {
    unsigned int five = 5;
    int negative_five = -5;
    if (negative_five < five) // несоответствие сравниваемых типов
        std::cout << "true";
    else
        std::cout << "false";
    return 0;
}
```

Она выведет "false". Вы стали жертвой неявного приведения. В ходе операции сравнения `negative_five` неявно преобразуется в целое число без знака. Происходит то, что называется расширением целочисленного представления. К сожалению, эта последовательность битов в форме дополнения до двух представляет огромное число, которое, конечно, значительно больше, чем `five`. Рекомендация «ES.100. Не смешивайте арифметику со знаком и без знака» прямо говорит о корне подобных бед.

Обратите внимание, что в примере используются явные типы, а не `auto`. Если бы переменные были объявлены с типом `auto`, то переменная `five` получила бы тип `int`. Чтобы присвоить ей целочисленный тип без знака, пришлось бы объявить ее как:

```
auto five = 5u;
```

По умолчанию присваивается тип со знаком. Это правильный выбор C++ по умолчанию.

Попытавшись скомпилировать этот код, вы почти наверняка столкнетесь в строке 5 с предупреждением, отмечающим несоответствие между типами со знаком и без знака. Вы имеете полное право написать этот код (это не ошибка), но могут возникнуть проблемы. Вот почему следует обращать внимание на все предупреждения и устранять их.

Проблема в том, что при смешивании арифметики со знаком и без знака могут произойти нежелательные и совершенно предсказуемые вещи. Значения со знаком будут преобразованы в значения без знака, и с этого момента любое сравнение сможет дать неверный результат. В больших проектах одна из ваших библиотек может экспортировать результаты без знака и заразить библиотеки, использующие только значения со знаком. Беда нависнет над вами.

Хуже того, некоторые проекты скрывают использование `unsigned`, определяя короткие псевдонимы, например:

```
using u32 = unsigned int;
```

Вид ключевого слова `unsigned` должен быть подобен мигающим неоновым огням, требующим нажать на тормоз и с особой осторожностью двигаться по дороге, это сигнал: впереди опасность.

Однако есть некоторые ситуации, когда применение `unsigned` является правильным выбором. Рекомендация «ES.101. Используйте беззнаковые

типы для манипуляций с битами» определяет одну из них. Однако это случается редко:

- при моделировании аппаратных регистров, которые содержат беззнаковые значения;
- при работе с размерами, а не с количествами, например со значениями, возвращаемыми `sizeof`;
- при выполнении манипуляций с битовыми масками, которые не участвуют в арифметических операциях.

Вот основное правило: при выполнении любых арифметических действий, включая сравнение, используйте тип со знаком. Если вы используете беззнаковый тип, только чтобы получить дополнительный бит в представлении, значит, вы выбрали его неправильно и следует использовать более широкий тип или признаться в применении очень рискованной оптимизации. Не будет удивительно, если в языке рано или поздно появится тип битового поля, что сделает даже выделение этого случая излишним.

При выполнении любых арифметических действий, включая сравнение, используйте тип со знаком. Если вы применяете беззнаковый тип, только чтобы получить дополнительный бит в представлении, значит, вы выбрали его неправильно и следует использовать более широкий тип или признаться в применении очень рискованной оптимизации.

К сожалению, в стандартной библиотеке есть довольно значительная ошибка. Все функции-члены контейнеров, возвращающие размер, возвращают беззнаковый тип `size_t`. Причина кроется в неправильном понимании разницы между количеством и объемом. Размер контейнера — это количество элементов, которые он содержит. Размер объекта — это объем памяти, который он занимает.

К счастью, начиная с C++20, мы были осчастливлены появлением функции `std::ssize` (сокращенно от `sign size` — «размер со знаком»). Она возвращает значение со знаком. Откажитесь от любого использования функции-члена `size` и применяйте `std::ssize`, не являющуюся членом, например:

```
auto collection = std::vector<int>{1, 3, 5, 7, 9};  
auto element_count = std::ssize(collection);
```

БУФЕРЫ И РАЗМЕРЫ

Коль скоро речь зашла о размерах, давайте рассмотрим буферы. Работая с буферами, следует помнить о двух важных аспектах: об адресе и длине буфера. Если буфер не переполняется, то все в порядке. Переполнение буфера — это класс ошибок времени выполнения, которые порой чертовски сложно обнаружить.

Например, взгляните на следующий фрагмент кода:

```
void fill_buffer(char* text, int length)
{
    char buf[256];
    strncpy(buf, text, length);
    ...
}
```

Очевидная ошибка заключается в том, что `length` может быть больше 256. Если не убедиться, что массив символов имеет достаточный размер, можно случайно переполнить его.

Обратите внимание, что `buf` имеет тип `char[256]`, который отличается, например, от типа `char[128]`. Размер важен, но его легко потерять, передав адрес начала массива в функцию, которая принимает только один указатель. Взгляните на следующую пару функций:

```
void fill_n_buffer(char*);  
  
void src(std::ifstream& file)  
{  
    char source[256] = {0};  
    ... // заполняет source содержимым файла  
    fill_n_buffer(source);  
}
```

`fill_n_buffer` ожидает аргумент `char*`, но ей передается `char[256]`. Это называется распадом массива (*array decay*), потому что фактический тип распадается в нечто менее полезное. Здесь информация о размере массива потерялась: тип `char[256]` распался до `char*`. Остается только надеяться, что `fill_n_buffer` сумеет воспользоваться этой сокращенной информацией. Присутствие `n` в имени может указывать на то, что функция ожидает строку с завершающим нулем в стиле стандартной библиотеки C, но мы надеемся, что вы понимаете рискованность такого предположения, которое может оказаться ошибочным.

Код работает на опасном уровне абстракции. Есть вероятность переполнения буфера, поэтому код небезопасен. Правильный подход — использовать абстракцию буфера вместо прямой записи или чтения из памяти. Доступно несколько таких абстракций: `std::string` — несколько тяжеловесный подход к обработке изменяющихся строк символов, зато предусмотренный стандартом, однако мы не будем подробно останавливаться на его природе. Если требуется просто прочитать буфер, то можно использовать более легковесную абстракцию: `std::string_view` — облегченную версию `std::string`, состоящую только из константных функций-членов и специальных функций (конструктор по умолчанию; конструкторы перемещения и копирования; операторы присваивания, перемещения и копирования; деструктор). Обычно он реализуется как указатель и размер. Создать экземпляр `std::string_view` можно из указателя и размера, или только указателя, или пары итераторов, что делает этот тип очень гибким и предпочтительным выбором для работы со строками, когда требуется доступ только для чтения.

Если ваш буфер содержит что-то другое, отличное от символьного типа, то можно использовать другие варианты. В C++20 в библиотеку был добавлен тип `std::span` — облегченная версия `std::vector`, также состоящая только из константных функций-членов. Эти два типа избавляют от необходимости определять функции с параметрами для передачи пар «указатель/размер», достаточно лишь завернуть их в абстракцию буфера, используя `std::span` или `std::string_view`.

ПОДВЕДЕМ ИТОГ

Писать безопасный код на C++ стало проще, чем когда-либо. И это неудивительно, потому что одной из целей C++ является безопасность, а через нее и защищенность. Можно предпочесть безопасность типов и использовать такие абстракции, как `span`, препятствующие переполнению буфера. Можно отказаться от беззнаковых типов для значений, участвующих в арифметических операциях, и избежать конфликтов представлений. Можно выполнять приведение типов только при взаимодействии со старыми или плохо написанными библиотеками и обратиться к использованию размеченных объединений вместо объединений в стиле С. Все это приводит к более безопасному, простому и понятному коду, гарантируя, что ваша программа в полной мере использует безопасность типов, обеспечиваемую C++.

ГЛАВА 5.2

P.10. Неизменяемые данные предпочтительнее изменяемых

НЕПРАВИЛЬНЫЕ ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

C++ — это язык с неправильными значениями по умолчанию. Например, объекты создаются изменяемыми, если они не объявлены со спецификатором `const`. Но почему это неправильно?

Есть две операции, которые можно делать с данными: читать их и записывать. В языках ассемблера эти операции также называются загрузкой и сохранением. Данные загружаются из памяти в регистры и сохраняются из регистров в память. Очень необычно сохранять значение в память, не загрузив его предварительно. Точно так же очень необычно выполнять запись в объект, не прочитав его перед этим. Однако в чтении из объекта без последующей записи в него нет ничего необычного.

Объекты только для чтения более распространены, чем объекты для чтения и записи, хотя последние сами по себе гораздо более распространены, чем объекты только для записи. При назначении свойств по умолчанию в языке должен быть реализован выбор наиболее распространенного варианта, поэтому мы утверждаем, что объекты должны быть неизменяемыми по умолчанию.

Это может иметь некоторые интересные побочные эффекты. Представьте, что в C++ нет ключевого слова `const` и очень широко используется ключевое слово `mutable`. Вы сможете писать такие функции:

```
void f1(std::string id)
{
    auto write = id_destination(); // write – неизменяемая переменная
    ...
}
```

В ходе разработки функции наступит момент, когда вам понадобится квалифицировать переменную `write` как изменяемую, чтобы выполнить запись в нее, и это нарушит ее неизменяемость.

Представьте, что по завершении функции вы обратили внимание, что `write` остается неизменяемой. Это сразу будет воспринято как ошибка: какой смысл давать объекту имя, такое как `write`, и никогда не изменять его?

Возьмите на вооружение этот прием поиска ошибок: просто объявляйте все переменные с квалификатором `const` и убирайте его, только когда нужно изменить объект. Вы можете периодически переквалифицировать все объекты в константные и проверять, действительно ли в них что-то записывается, действительно ли добавление квалификатора `const` вызывает ошибку компилятора. Это приятный выигрыш, способствующий ясности кода.

Однако это не главное преимущество неизменяемых данных.

У меня есть машина¹ и солнцезащитные очки, которые я надеваю в машине. Они никогда не покидают салона машины. Находясь вне машины, я надеваю другие солнцезащитные очки. Мне трудно водить без солнцезащитных очков, когда солнце бьет в глаза, поэтому я стараюсь надевать их всегда, когда это необходимо. Соответственно, солнцезащитные очки находятся в одном из двух мест: у меня на переносице или в специальном кармане в дверце машины. Я приучил себя проверять, находятся ли они там, где должны быть, когда открываю дверцу машины. Мне действительно не нужно думать об этом.

У меня, к сожалению, не электромобиль. Надеюсь, это будет мой последний пожиратель бензина. Начиная поездку, я должен проверить, достаточно ли топлива в баке. Производители сделали все, чтобы упростить эту задачу. На приборной панели есть указатель уровня топлива со счетчиком, показывающим приблизительный запас хода, взглянув на который я могу узнать, как далеко можно уехать, прежде чем должен буду дозаправиться. Счетчик подсчитывает расстояние с определенным запасом, поэтому, умев выполнять простые арифметические действия, я обычно не оказываюсь на дороге с пустым баком. И все же мне приходилось попадать в сложные ситуации. В Великобритании есть длинные участки дорог, где нет заправочных станций, например, участки, проходящие через болота. Пробки в такой ситуации иногда заставляли меня поволноваться, если перед этим я давно не доливал топливо в бак.

¹ Рассказ Дж. Гая Дэвидсона.

314 Часть V. Пишите хороший код по умолчанию

Мне гораздо легче рассуждать о солнцезащитных очках, всегда находящихся в одном месте, чем о топливном баке с меняющимся уровнем топлива.

Взгляните на следующую функцию:

```
double gomez_radius(double cr, double x1, double x2,
                     double y1, double y2, double rh)
{
    assert(rh > 1.0f);
    auto lt1 = lovage_trychter_number(cr, x1, y1);
    auto lt2 = lovage_trychter_number(cr, x2, y2);
    auto exp = 1;
    while (lt1 > lt2) {
        lt2 *= rh;
        ++exp;
    }
    auto hc = haldwell_correction(cr, lt1, lt2);
    auto gr = (lt1 * lt2 * sqrt(hc)) / rh * exp;
    return gr;
}
```

Представьте, что эта функция скопирована прямо из учебника. Понять ее будет проще, если добавить квалификатор `const`:

```
double gomez_radius(double cr, double x1, double x2,
                     double y1, double y2, double rh)
{
    assert(rh > 1.0f);
    auto const lt1 = lovage_trychter_number(cr, x1, y1);
    auto lt2 = lovage_trychter_number(cr, x2, y2);
    auto exp = 1;
    while (lt1 > lt2) {
        lt2 *= rh;
        ++exp;
    }
    auto const hc = haldwell_correction(cr, lt1, lt2);
    auto const gr = (lt1 * lt2 * sqrt(hc)) / rh * exp;
    return gr;
}
```

Количество подвижных частей уменьшилось до двух: второе число `lt2` и `exp`.

Представьте, насколько сложной стала бы архитектура, если бы ускорение свободного падения значительно различалось по всей планете. А вспомните достижения в науке XX века, появившиеся после того, как скорость света была признана константой. Константы — хорошая штука! Их инвариантный характер помогает уменьшить количество тем для обдумывания. Там, где можно зафиксировать какое-либо значение, вы должны это сделать.

CONST В ОБЪЯВЛЕНИЯХ ФУНКЦИЙ

Мы не можем неявно сделать объекты константными по умолчанию, но можем использовать квалификатор `const`, создавая их. Функции-члены по умолчанию тоже считаются изменяемыми (способными изменять переменные-члены). Однако было бы разумнее квалифицировать их как константные до тех пор, пока им не понадобится что-то изменить.

Впрочем, ситуация с функциями немного сложнее. Как обсуждалось в главе 3.4, существует два вида константности: логическая и побитовая. Когда возникает необходимость что-то изменить внутри константной функции, ответьте на вопрос: как лучше поступить — убрать квалификатор `const` из объявления функции или снабдить переменные-члены квалификатором `mutable` и освободить их от правила, согласно которому переменные-члены не могут изменяться константной функцией-членом?

Это вопрос дизайна API. Цель квалификатора `const` — сообщить пользователю: «После вызова константной функции видимое состояние объекта не изменится». Если ваша функция на самом деле изменяет состояние абстракции, а не частные детали реализации, то ее не следует квалифицировать как `const`.

Например, представьте, что вы разрабатываете потокобезопасную очередь для передачи сообщений. Это обычное упражнение в курсе изучения безопасности потоков выполнения: для синхронизации потоков вместо общей памяти безопаснее использовать обмен сообщениями. Здесь уместно вспомнить максиму, упомянутую выше: «Не общайтесь, делясь памятью; делитесь памятью, общаясь». Впрочем, вы можете возразить, утверждая, что обмен сообщениями тоже является совместным использованием общей памяти.

Очередь передачи сообщений может иметь такой открытый интерфейс:

```
template <typename T>
class message_queue {
public:
    ... // определения для оператора &c.
    void push(T);
    T pop();
    const_iterator find(const T&) const;

private:
    mutable std::mutex m_lock;
    ... // другие детали реализации
};
```

Как правило, очередью пользуются два потока: производитель и потребитель. Производитель отправляет, а потребитель извлекает сообщения. Функции `push` и `pop`, безусловно, изменяют объект, но функция `find`, отыскивающая конкретный объект в очереди, этого делать не должна. Однако на время поиска этой функции потребуется заблокировать мьютекс, то есть выполнить неконстантную операцию. Если бы `find` не была константной функцией, то она могла бы изменить любую переменную-член. Но она определена как константная, поэтому единственное значение, которое она может изменить, — мьютекс, отмеченный квалификатором `mutable`. Это видно уже из объявления класса. И действительно, единственный контекст, где член-мьютекс не требуется объявлять изменяемым, — это класс, не имеющий константных функций-членов. Однако даже в этом случае объявление мьютексов изменяемыми считается хорошей привычкой.

Параметры функции тоже считаются изменяемыми по умолчанию. Они действуют подобно локальным переменным. В функции обычно не принято изменять аргументы: они используются для настройки ее действий. Заметным исключением являются функции из `<algorithm>`, принимающие итераторы. Однако, учитывая, что параметры функции в любом случае редко объявляются константными, вы можете решить отказаться от добавления квалификатора `const` в определения параметров функций. Например, объявления:

```
void f(char const* const p);
void g(int const i);
```

могут показаться скорее излишне педантичными, чем правильными. Это вопрос стиля и вкуса.

Есть еще один вопрос стиля и вкуса, который стоит рассмотреть, он касается квалификатора `const`.

- Ссылки не требуется снабжать квалификатором `const`.
- Указатели снабжаются квалификатором `const` справа.
- Функции-члены снабжаются квалификатором `const` справа.
- Типы могут снабжаться квалификатором `const` и справа, и слева от типа. Эти способы известны как East `const` (восточный `const`) и `const` West (западный `const`).

Ссылки не требуется снабжать квалификатором `const`, потому что их нельзя переадресовать; то есть нельзя изменить адрес, на который указывает ссылка. Добавление квалификатора `const` не добавляет дополнительной информации.

Указатели должны снабжаться квалификатором `const` справа от типа, иначе возникает вероятность неоднозначного толкования:

```
int const * j; // Что квалифицируется как константа?  
               // Значение int или указатель?
```

То же верно в отношении функций-членов:

```
int const my_type::f1(); // Что квалифицируется как константа?  
                       // Возвращаемый тип или функция-член?
```

Это относится только к типам, допускающим выбор местоположения квалификатора `const`. Возможно, нет смысла быть догматичными в отношении выбора стороны, где должен находиться квалификатор `const`, однако некоторые предпочитают восточный `const`, как вы, возможно, поняли из вышеизложенного. Им нравится непротиворечивость такого способа, а кроме того, подобные объявления кажутся чуть более естественными:

```
int const& a = ...; // a – ссылка на константу int  
int const* b;       // b – указатель на константу int  
int & c = ...;      // c – ссылка на int  
int const d;        // d – константа типа int
```

Мало чем поможет в выработке стиля при расположении квалификатора `const` и знание английского языка, потому что в нем принято помещать прилагательные слева от определяемого существительного, так что акцентируется определенное противоречие в написании `const` справа. С другой стороны, согласованность может способствовать читаемости, но это не жесткое правило. Даже редакторы в издательстве, работая с текстами, подобными тексту этой книги, отмечают, что благополучно научились воспринимать `int` как прилагательное, описывающее природу имени, так же как `const`. Порядок прилагательных – интересное отступление (почему мы говорим «большая дружелюбная собака», а не «дружелюбная большая собака»?), но оно выходит за рамки нашего обсуждения. В конечном счете удобочитаемость предпочтительнее согласованности, а более широком смысле pragmatizm предпочтительнее догматизма, особенно в инженерии.

Кроме параметров, стоит обдумать и возвращаемые значения. Значение, возвращаемое функцией, является правосторонним (rvalue). Нет смысла квалифицировать его как `const`, потому что оно будет присвоено и далее использовано при создании объекта или уничтожено. Есть ли во всем этом хоть какая-то причина квалифицировать возвращаемое значение как `const`?

Этот наводящий вопрос должен натолкнуть вас на мысль, что на самом деле причина есть. Возвращаемый объект действительно является правосторонним значением, но он не уничтожается сразу и может использоваться для вызова функций. Взгляните на следующий класс:

```
template <typename T>
class my_container {
public:
    ...
    T operator[](size_t);
    T const operator[](size_t) const;
};
```

Оператор индекса, или, точнее, оператор квадратных скобок, обычно используется для получения объекта по значению. Однако у этого контейнера могут быть очень нестабильные итераторы, которые становятся недействительными из-за операций в другом потоке. В таком случае возврат по значению является единственным способом безопасного экспорта значений. В классе определены две перегруженные версии этого оператора, чтобы дать возможность выбирать правильную версию [] для константных и неконстантных объектов. Взгляните на следующий фрагмент кода:

```
my_container<Proxies> const p = {a, b, c, d, e};
p[2].do_stuff();
```

При наличии перегруженной версии функции-члена `Proxies::do_stuff()` с квалификатором `const` для rvalue, возвращаемого вызовом `p[2]`, будет вызвана правильная версия функции.

Вы, наверное, думаете про себя, что этот пример далек от реальности, и мы согласимся с вами. Сама эта возможность показывает, что квалификация возвращаемых типов ключевым словом `const` должна быть обдуманным и осознанным шагом, так как сам этот шаг по своей сути необычен.

Наконец, есть исключение при определении параметров функции как констант, когда они передаются по указателю или по ссылке. По умолчанию такие аргументы должны приниматься по ссылке на константу или

по указателю на константу. Это сигнализирует вызывающему коду, что функция не будет изменять передаваемые ей объекты, и тем самым подкрепляет идею входных и входных-выходных параметров, обсуждавшуюся в главе 4.1. Здесь отделение типа от квалификатора `const` для ссылки и указателя становится полезнее. Например:

```
int f1(std::vector<int> const& a, int b, std::vector<int> & c);
```

Ясно видно, что функция использует два входных параметра, `a` и `b`, и изменяет третий, выходной параметр `c` (подумайте, какие значения могут быть изменены и возвращены в точку вызова). Возвращаемое значение, вероятно, определяет код ошибки. Отделение `const&` от типа идентификатора подчеркивает квалификацию, что дает ожидания о роли объекта в функции.

ПОДВЕДЕМ ИТОГ

Заманчиво было бы отреагировать на эту рекомендацию, разбросав `const` по всему коду и пробормотав себе под нос: «Готово». Однако все не так просто. Назначение квалификатора `const` по умолчанию не всегда верно. Конечно, в большинстве случаев объекты действительно можно и нужно создавать с квалификатором `const`, а затем изменять их квалификацию, если понадобится, но это вопрос не элементарного механического добавления/удаления ключевого слова `const` рядом с типом: в некоторых местах квалификатор `const` действительно неуместен. А в целом направление мысли в этой рекомендации абсолютно верное.

- Опираясь на неизменяемые данные, проще рассуждать.
- Объявляйте объекты и функции-члены константными везде, где это возможно.
- Оцените, какой стиль `const` ближе вам — восточный или западный.
- Параметры функции, передаваемые и возвращаемые по значению, не выигрывают от применения квалификатора `const`, зато такой подход обеспечивает правильное распространение константности в некоторых конкретных ситуациях.

ГЛАВА 5.3

I.30. Инкапсулируйте нарушения правил

СОКРЫТИЕ НЕПРИГЛЯДНЫХ ВЕЩЕЙ

В моем доме есть комната, куда я никого не пускаю¹. Она слишком мала для спальни и слишком велика для гардеробной, но недостаточно велика, чтобы ее можно было использовать как кабинет. Я понятия не имею, что замышлял архитектор, но я нашел свое применение этой комнате.

Я люблю, чтобы все было чисто и аккуратно и чтобы все вещи лежали на своих местах. Так мне проще находить вещи, когда они мне нужны, не тратя времени на поиск.

К сожалению, Вселенная не всегда поддерживает меня в моих стремлениях. Я тщательно сортирую вещи по категориям, чтобы похожие предметы хранились вместе, но у меня может быть излишек, например, степлеров, дыроколов или приспособлений для удаления скобок, потому что я стар и все еще живу в мире ручек, скобок и папок со скоросшивателями формата А4.

Лишние предметы отправляются в эту комнату.

Недостаток моей системы в том, что она не предусматривает разумного способа хранения лишних степлеров. Я просто собираю их все в одном месте и надеюсь на лучшее. Я не буду вдаваться в подробности, почему у меня появляются лишние степлеры и почему я их храню, а не выбрасываю. Достаточно сказать, что у меня есть небольшая, скрытая от посторонних глаз комната, наполненная хламом. Можно даже сказать: плотно набитая.

К чему я веду: общая проблема в том, что иногда, несмотря на наши самые лучшие намерения, события разворачиваются не по плану.

¹ Житейский и концептуальный опыт одного из авторов, Дж. Гая Дэвидсона.

Инженеры живут и работают в реальном мире, и в этом мире они сталкиваются с живыми существами, полными противоречивых идей и желаний. Самые лучшие принципы и самые лучшие намерения могут рухнуть, столкнувшись с неизбежным вторжением с более низкого уровня абстракции. Эта рекомендация призывает свести к минимуму видимость, влияние и последствия таких ужасов.

Вспомните пример кода в главе 2.1, где на задаче синтаксического анализа файла параметров было показано, как ситуация может выйти из-под контроля из-за расплазания области видимости. Решение заключалось в тщательном разделении задач, их абстрагировании за хорошо продуманными интерфейсами. Получившееся решение могло выполнять синтаксический анализ любого источника параметров, который можно представить в виде потока данных. Ну а что, если бы эта возможность была включена в предварительные требования?

В описании к этой рекомендации в Core Guidelines представлен аналогичный пример. В нем программе сообщается, откуда она может получить входные данные, и та должна проанализировать соответствующий поток. Обычно программа запускается из командной строки и принимает до двух параметров. Возможны три источника входных данных: стандартный ввод, командная строка и внешний файл. Как следствие, первый параметр идентифицирует источник ввода, например 0, 1 или 2, а второй определяет команду или имя файла. Соответственно, возможны три варианта командной строки:

| | |
|------------------------|--|
| sample.exe 0 | Прочитать команды из стандартного ввода |
| sample.exe 1 help | Выполнить команду help |
| sample.exe 2 setup.txt | Выполнить команды, перечисленные в setup.txt |

Воспроизведем слегка измененный код примера из главы 2.1:

```
enum class input_source { std_in, command_line, file };
```

```
bool owned;
std::istream* input;

switch (source) {
case input_source::std_in:
    owned = false;
    input = &std::cin;
    break;
case input_source::command_line:
    owned = true;
    input = new std::istringstream{argv[2]};
    break;
}
```

```

case input_source::file:
    owned = true;
    input = new std::ifstream{&argv[2]};
    break;
}
std::istream& in = *input;

```

Верху определяются: перечисление с именем `input_source`, описывающее три возможных источника данных, логическое значение с именем `owned` и `std::istream` с именем `input`. Предположим, что переменная `source` инициализируется значением первого параметра командной строки: операторы `case` сравнивают его с членами перечисления `input_source` и присваивают переменной `input` указатель либо на существующий объект `std::istream` — `std::cin`, либо на новый объект `std::ifstream` или `std::istringstream`. В результате возникает проблема: в каких-то случаях мы должны уничтожить поток перед завершением программы, а в каких-то нет. Определить необходимость удаления потока статически невозможно, поэтому необходим флаг, сигнализирующий о том, нужно ли уничтожать поток. Если флаг имеет значение `true`, то поток должен быть уничтожен.

Этот код не соответствует рекомендациям из Core Guidelines. Например, «ES.20. Всегда инициализируйте объект» не предполагает никакой двусмысленности, а ее нарушение часто приводит к нежелательным результатам. В этом примере переменные `owned` и `input` не инициализируются. Автор не захотел присваивать им начальные значения, чтобы тут же изменить их, прежде чем они будут использованы.

Как инкапсулировать такое нарушение правил?

ПОДДЕРЖАНИЕ ВИДИМОСТИ, ЧТО ВСЕ В ПОРЯДКЕ

Мы уже познакомились с шаблоном немедленно вызываемого лямбда-выражения (Immediately Invoked Lambda Expression, IILE), поэтому попробуем применить его:

```

auto input = [&]() -> std::pair<bool, std::istream*> {
    auto source = input_source_from_ptr(argv[1]);
    switch (source) {
        case input_source::std_in:

```

```

    return {false, &std::cin};
case input_source::command_line:
    return {true, new std::istringstream{argv[2]}};
case input_source::file:
    return {true, new std::ifstream{argv[2]}};
}
}();

```

Для знакомых с идиомой это заметное улучшение.

Однако здесь игнорируется еще одно правило, а именно «I.11. Никогда не передавайте владение через простой указатель (`T*`) или ссылку (`T&`)». Проблема очевидна: автор должен помнить о принадлежности потока данных функции, то есть он не должен пытаться уничтожить поток стандартного ввода. Простой указатель не содержит информации о владении, поэтому необходим дополнительный признак, сообщающий о принадлежности объекта. Кроме того, автор должен не забыть написать:

```
if (input.first) delete input.second;
```

в каждом пути выполнения кода. Это похоже на идиому получения ресурсов при инициализации (Resource Acquisition Is Initialization, RAII)¹, а ее можно смоделировать с помощью класса.

Назовем класс `command_stream`. При необходимости он должен владеть потоком `std::istream`, поэтому начнем с этого:

```

class command_stream {
private:
    bool owned;           // Может владеть потоком std::istream
    std::istream* inp;   // Это сам поток данных std::istream
};

```

Здесь нет ничего примечательного. Деструктор реализуется просто:

```

class command_stream {
public:
    ~command_stream() {
        if (owned) delete inp;
    }

private:
    bool owned;           // Может владеть потоком std::istream
    std::istream* inp;   // Это сам поток данных std::istream
};

```

¹ Обсуждается в главе 5.6.

Конструктор должен принимать параметр для индикации того, к какому потоку входных данных должен быть перенаправлен, и необязательное имя файла или команду. К счастью, у нас уже есть готовое перечисление, поэтому класс теперь выглядит так:

```
class command_stream {
public:
    command_stream(input_source source, std::string token) {
        switch (source) {
            case input_source::std_in:
                owned = false;
                inp = &std::cin;
                return;
            case input_source::command_line:
                owned = true;
                inp = new std::istringstream{ token };
                return;
            case input_source::file:
                owned = true;
                inp = new std::ifstream{ token };
                return;
        }
    }

    ~command_stream() {
        if (owned) delete inp;
    }

private:
    bool owned;           // Может владеть потоком std::istream
    std::istream* inp;   // Это сам поток данных std::istream
};
```

Однако, похоже, мы связали перечисление с классом `command_stream`. Но так ли это необходимо? Всегда стоит потратить немного времени, чтобы как можно раньше отвязаться от подобных зависимостей.

Совсем не обязательно импортировать перечисление `input_source` в класс. Можно просто определить три конструктора. В простейшем случае `std::istream` — это `std::cin`. В этой ситуации ничего не нужно создавать и нет проблем с владением. Мы можем сделать этот конструктор конструктором по умолчанию:

```
class command_stream {
public:
    command_stream()
        : owned(false)
        , inp(&std::cin) {}
```

```

~command_stream() {
    if (owned) delete inp;
}

private:
    bool owned;           // Может владеть потоком std::istream
    std::istream* inp;   // Это сам поток данных std::istream
};

```

На самом деле можно поступить еще лучше и использовать инициализацию членов значениями по умолчанию:

```

class command_stream {
public:
    command_stream() = default;
    ~command_stream() {
        if (owned) delete inp;
    }
private:
    bool owned = false;           // Может владеть потоком std::istream
    std::istream* inp = &std::cin; // Это сам поток данных std::istream
};

```

Два других конструктора должны принимать `std::string` и ничего больше, поэтому их нужно как-то различать. Есть несколько способов сделать это, но мы используем тег.

Тег — это структура без членов. Она дает возможность определить переопределенные версии функций, потому что перегрузка определяется типами параметров. Итак, определим тег `from_command_line`, с помощью которого будем отличать два оставшихся конструктора:

```

class command_stream {
public:
    struct from_command_line {};
    command_stream() = default;

    command_stream(std::string filename)
        : owned(true)
        , inp(new std::ifstream(filename))
    {}

    command_stream(std::string command_list, from_command_line)
        : owned(true)
        , inp(new std::istringstream(command_list))
    {}

    ~command_stream() {
        if (owned) delete inp;
    }
}

```

```
private:
    bool owned = false; // Может владеть потоком std::istream
    std::istream* inp = &std::cin; // Это сам поток данных std::istream
};
```

Наконец, нам нужно, чтобы класс вел себя как `std::istream`, а это означает наличие оператора преобразования:

```
class command_stream {
public:
    struct from_command_line {};
    command_stream() = default;

    command_stream(std::string filename)
        : owned(true)
        , inp(new std::ifstream(filename))
    {}

    command_stream(std::string command_list, from_command_line)
        : owned(true)
        , inp(new std::istringstream(command_list))
    {}

    ~command_stream() {
        if (owned) delete inp;
    }

    operator std::istream&() { return *inp; }

private:
    bool owned = false; // Может владеть потоком std::istream
    std::istream* inp = &std::cin; // Это сам поток данных std::istream
};
```

Вот и все. Все детали аккуратно спрятаны за общедоступным интерфейсом. Переменные `owned` и `inp` по умолчанию получают значения `false` и `std::cin` соответственно. Они остаются неизменными, когда экземпляр инициализируется конструктором по умолчанию. Другие конструкторы используются для инициализации строковым или файловым потоком. Теперь можно переписать наш фрагмент кода:

```
auto input = [&]() -> command_stream {
    auto source = input_source_from_ptr(argv[1]);
    switch (source) {
        case input_source::std_in:
            return {};
        case input_source::command_line:
            return {{argv[2]}}, command_stream::from_command_line{};
    }
}
```

```
case input_source::file:  
    return {argv[2]};  
}  
};
```

Код получился намного чище, чем прежний. Оператор преобразования позволяет обращаться с объектом так, будто он имеет тип `std::istream&`, благодаря чему можно продолжать пользоваться знакомым и удобным синтаксисом с применением перегруженных операторов шевронов (`>>` и `<<`). При выходе имени за пределы области видимости объект `command_stream` будет автоматически уничтожаться, а флаг владения обеспечит надлежащее уничтожение объекта `std::istream`. Это довольно сложная часть реализации, но она абстрагируется за простым интерфейсом.

ПОДВЕДЕМ ИТОГ

Надеемся, вы заметили, что абстрагирование является постоянной темой всей этой книги: абстракции помогают локализовать и минимизировать сложности. В главе 2.1 было показано, что абстракция — это нечто большее, чем просто инкапсуляция и скрытие данных: она позволяет скрывать сложности во внутренней реализации и экспорттировать только самый минимум, необходимый для взаимодействия с объектом.

В главе 2.2 демонстрировалось, как может увеличиваться число аргументов, и вы могли заметить, что это часто связано с отсутствием абстракции. Сложность понимания многих параметров скрыта за абстракцией, описывающей их роли.

В главе 4.1 мы выполнили аналогичное преобразование и собрали беспорядочную кучу данных в единую абстракцию возвращаемого значения вместо организации приема нескольких выходных параметров.

Core Guidelines в некоторых случаях явно говорит о необходимости абстрагирования. Выполняя рекомендацию «С.8. Используйте класс, а не структуру, если какой-то член не является общедоступным», программисты дают понять: что-то абстрагируется. По умолчанию члены класса закрыты, а это сигнализирует вам, что есть что-то, что видеть совсем не обязательно.

Рекомендация «ES.1. Если есть возможность, используйте стандартную библиотеку вместо других библиотек и кода, написанного вручную» явно

говорит, что код в стандартной библиотеке, как правило, имеет более высокий уровень абстракции. Как уже отмечалось выше, иногда разработчики создают код, в котором просто переписывают `std::rotate` или `std::mismatch`, не осознавая этого, и тем самым ухудшают свою работу.

Рекомендация «ES.2. Если есть возможность, используйте подходящие абстракции вместо языковых особенностей» явно предлагает использовать абстракции. Она наглядно показывает, что многие прикладные идеи часто дальше от чистого языка, чем от соответствующих комбинаций библиотечных функций.

Даже термин «абстрактный базовый класс» описывает объект, скрывающий сложность и представляющий только общедоступный API, изолируя клиента от тонкостей реализации. Абстракция лежит в основе разработки программного обеспечения на C++.

Абстракции помогают локализовать и минимизировать сложность.

Когда нарушение рекомендаций из Core Guideline кажется неизбежным, старайтесь свести к минимуму и скрыть это нарушение. Одним из основных принципов C++ является абстракция с нулевой стоимостью, можно использовать ее, чтобы максимально повысить ясность и понятность вашего кода.

ГЛАВА 5.4

ES.22. Не объявляйте переменные, пока не получите значения для их инициализации

ВАЖНОСТЬ ВЫРАЖЕНИЙ И ОПЕРАТОРОВ

Это четвертая включенная в эту книгу рекомендация из раздела Expressions and Statements («Выражения и инструкции») в Core Guidelines. Одна из причин, почему так много внимания уделяется этому разделу, в том, что он очень большой (включает более 60 рекомендаций), и к тому же, самое главное, он затрагивает суть C++. Раздел открывается следующим предложением: «Выражения и инструкции — это самый низкоуровневый и самый прямой способ выражения действий и вычислений».

Текущая рекомендация прямо связана с «ES.5. Минимизируйте области видимости» и «ES.10. Объявляйте имена по одному в каждом объявлении». Как мы показали в обсуждении рекомендации ES.5, небольшие размеры областей видимости создают контекст и тем самым значительно улучшают читаемость кода. Они также упрощают управление ресурсами, потому что их освобождение сопровождается детерминированным уничтожением, что сводит к минимуму вероятность их существования дольше необходимого. В обсуждении рекомендации ES.10 мы показали преимущества разделения объявлений вместо группировки в стиле C. Возможность группировки объявлений с одним типом является лишь средством поддержки обратной совместимости и не более того. Она не дает никаких преимуществ программисту на C++ и на самом деле может запутать и усложнить код.

Еще один способ улучшить читаемость — отложить объявление объектов до самого последнего момента. Функции растут и меняются. Такова их природа. Часто случается, что мы не сразу замечаем их безудержное разрастание и тогда получаем довольно громоздкий набор объектов и логики. Осознав произошедшее, мы пытаемся разделить функцию, абстрагировать ее на более мелкие функции, скрыть сложность и восстановить порядок в исходном коде. Проделать это трудно, если объявление и инициализация находятся в разных строках, особенно если они располагаются не рядом друг с другом. Давайте совершим небольшое путешествие по трем стилям программирования.

ОБЪЯВЛЕНИЕ В СТИЛЕ С

Раньше язык C требовал, чтобы все переменные объявлялись в начале функции до любого выполняемого кода. Это провоцировало естественное и непреднамеренное стремление объявлять больше переменных, чем нужно. По мере изменения алгоритма и целей функции менялся и контекст, необходимый для ее выполнения, и, как следствие, некоторые переменные оказывались ненужными. Например:

```
int analyze_scatter_data(scatter_context* sc) {
    int range_floor;
    int range_ceiling;
    float discriminator;
    int distribution_median;
    int distribution_lambda;
    int poisson_fudge;
    int range_average;
    int range_max;
    unsigned short noise_correction;
    char chart_name[100];
    int distribution_mode;
    ... // и т. д.
}
```

Первая версия этой функции была написана десять лет назад и с тех пор претерпела множество изменений. Пять лет назад переменную `distribution_median`, накапливающую суммарное значение, заменила `distribution_mode`, но обе они продолжают вычисляться в коде, даже притом, что медиана стала больше не нужна. Никто не удалил состояние, объявленное для поддержки неиспользуемой части алгоритма. С глаз долой — из головы вон. Кроме того, у инженеров есть пагубная привычка добавлять новые переменные

в конец списка, а не искать им правильное местоположение в этом списке, поэтому `range_average` и `range_max` оказались отделены в нем от `range_floor` и `range_ceiling`.

Если бы программист был целеустремленным, увлеченным и не торопился, то он убрал бы эти переменные из функции, чем улучшил ее ясность, а остальные собрал бы вместе, чтобы показать, как связаны между собой части состояния. Если бы у программиста был под рукой Lint, инструмент статического анализа кода, выпущенный для широкого использования в 1979 году, тот бы предупредил о неиспользуемых объектах и облегчил программисту эту задачу. Но это не случилось, и функция занимает несколько лишних, неиспользуемых байтов в стеке. В настоящее время это прошло бы незамеченным, потому что компиляторы удаляют неиспользуемые переменные согласно правилу «как если бы», но в условиях ограниченных вычислительных ресурсов 1970-х годов, когда размер файлов мог превысить объем ОЗУ компьютера, это имело существенное значение.

Еще одна похожая проблема заключалась в повторном использовании переменных. Функция могла повторно использовать целочисленную переменную в качестве счетчика циклов или для сохранения результатов вызовов функций. Как следствие — если функция изменялась, повторно используемые переменные могли приобретать неожиданные новые значения, и их повторное использование грозило оказаться недопустимым. Из-за ограниченности ресурсов приходилось также жертвовать удобочитаемостью кода.

Еще один отрицательный эффект раннего объявления — некоторая беспорядочность инициализации. Легко было создать зависимость между переменными и использовать их до инициализации. Каждую функцию требовалось тщательно изучить на наличие связей между переменными. Всякий раз, когда строка кода перемещалась в другое место, было важно обеспечить соблюдение всех зависимостей.

Это не самый приятный стиль разработки. Объявление пакета переменных, а затем их тщательная инициализация друг за другом — сложный процесс, чреватый ошибками и неприятный до зубовного скрежета. В C++ появилась поддержка позднего объявления, и первое, что произошло, — структуры и встроенные типы стали объявляться и инициализироваться в безопасном порядке. Это был значительный рывок вперед. Теперь можно не беспокоиться о готовности структуры к инициализации, а просто объявить все ее зависимости, инициализировать их, а затем безопасно инициализировать структуру, не заботясь ни о чем.

ОБЪЯВЛЕНИЕ С ПОСЛЕДУЮЩЕЙ ИНИЦИАЛИЗАЦИЕЙ

На ранних стадиях развития C++ код все еще выглядел будто написанный на C, хотя C++ предлагал новый стиль программирования. Можно было увидеть, например, такой код:

```
class Big_type {
    int m_acc;
    int m_bell;
    int m_length;

public:
    Big_type();
    void set_acc(int);
    void set_bell(int);
    void set_length(int);
    int get_acc() const;
    int get_bell() const;
    int get_length() const;
};

void fn(int a, int b, char* c)
{
    Big_type bt1;
    bt1.set_acc(a);
    bt1.set_bell(b);
    bt1.set_length(strlen(c));
    ...
    Big_type bt2;
    bt2.set_acc(bt1.get_bell() * bt1.get_acc());
    bt1.set_bell(bt2.get_acc() * 12);
    ...
}
```

В этом коде есть кое-что хорошее. Данные заключены в приватный интерфейс, переменные в функции объявляются по мере необходимости, но функции *get* и *set* раздражают. Зачем утруждать себя размещением деталей реализации в приватном интерфейсе только для того, чтобы открыть к ним доступ через функции *get* и *set*? Вдобавок зачем размещать приватный интерфейс в верхней части определения, если большинство клиентов интересует только общедоступный интерфейс?

Мы коснулись этой темы в обсуждении рекомендации «C.131. Избегайте тривиальных геттеров и сеттеров»: это пережиток времен парадигмы «объявление с последующей инициализацией». Конструктор, если он вообще

существует, скорее всего, установит все значения равными нулю. Иногда приходится сталкиваться с правилами программирования, такими как «Убедитесь, что для каждой переменной-члена есть геттеры и сеттеры», принятыми в отдельных организациях для поддержки такого стиля программирования. Даже в начале 2000-х еще приходилось наблюдать нежелание объявлять конструкторы из-за связанных с этим накладных расходов на инициализацию всех переменных-членов нулевыми значениями с последующей их перезаписью. И только когда потребовался строгий детерминизм, программисты в большинстве своем стали писать и использовать конструкторы.

Мы, конечно, говорим о конструкторах по умолчанию, с помощью которых разработчик класса может диктовать начальные значения (обычно 0). Большой сдвиг произошел, когда программисты начали использовать конструкторы не по умолчанию и даже полностью исключать конструкторы по умолчанию. Это третий стиль программирования.

МАКСИМАЛЬНОЕ ОТКЛАДЫВАНИЕ ОБЪЯВЛЕНИЯ

Конструкторы по умолчанию имеют свою нишу, но они предполагают существование значения по умолчанию. Некоторые ранние реализации C++ требовали наличия в классах конструкторов по умолчанию, если их экземпляр предполагалось хранить в `std::vector`. Это требование распространялось на переменные-члены, типы которых тоже должны были иметь значения по умолчанию, или чтобы конструктор по умолчанию мог создавать их с осмысленными значениями. К счастью, это требование осталось в прошлом, и мы теперь советуем добавлять конструкторы по умолчанию с осторожностью.

Возможно, вам интересно, как это связано с обсуждаемой рекомендацией. Все просто. Если конструктор требует выполнения полного набора начальных условий, то вы не сможете создать экземпляр класса, пока не будете готовы его использовать. Как можно видеть из примеров прежних стилей инициализации, «объявление с последующей инициализацией» в любой форме является случайностью и непредсказуемостью, ожидающей своего часа, поскольку нет ничего, что могло бы свидетельствовать о готовности объекта к использованию. В ходе инициализации могут добавиться дополнительные инструкции, что еще больше запутает развитие состояния

в вашей функции. Рассуждения о готовности или неготовности объектов к использованию — ненужное бремя, которое можно убрать, потребовав полной инициализации в момент объявления. Давайте модифицируем класс `Big_type`:

```
class Big_type {
    int m_acc;
    int m_bell;
    int m_length;

public:
    Big_type(int acc, int bell, int length);
    void set_acc(int);
    void set_bell(int);
    void set_length(int);
    int get_acc() const;
    int get_bell() const;
    int get_length() const;
};

void fn(int a, int b, char* c)
{
    Big_type bt1(a, b, strlen(c));
    ...
    Big_type bt2(bt1.get_bell() * bt1.get_acc(), bt2.get_acc() * 12, 0);
    ...
}
```

Оба объекта, `bt1` и `bt2`, теперь можно объявить константными, что предпочтительнее изменяемого состояния. Сеттеры сохранились в классе, но вполне вероятно, что теперь они не нужны. Код стал более понятным.

Есть еще одна причина отложить объявление до первого использования — устранение избыточности. Взгляните на следующий код:

```
class my_special_type {
public:
    my_special_type(int, int);
    int magical_invocation(float);
    ...
};

int f1(int a, int b, float c) {
    my_special_type m{a, b};
    if (a > b) return a;
    prepare_for_invocation();
    return m.magical_invocation(c);
}
```

Очевидно, что нет необходимости объявлять `m`, пока не будет вызвана функция `prepare_for_invocation()`. На самом деле здесь вообще нет необходимости объявлять именованное значение. Если переписать эту функцию, соблюдая рекомендацию «ES.5. Минимизируйте области видимости», то можно прийти к следующему:

```
int f2(int a, int b, float c) {
    if (a > b) return;
    prepare_for_invocation();
    return my_special_type{a, b}.magical_invocation(c);
}
```

Скорее всего, компилятор сгенерирует идентичный ассемблерный код согласно правилу «как если бы» и при отсутствии побочных эффектов в конструкторе, поэтому здесь мы не получаем оптимизации производительности. И все же объем кода стал меньше на одну строку, а также исчезла возможность запутать код между объявлением и использованием экземпляра `my_special_type`.

Обратите внимание, на сколько мы отложили создание экземпляра. Сначала мы имели левостороннее значение (`lvalue`) с именем `m` в начале функции, а теперь получили правостороннее значение (`rvalue`) в конце. Новая версия функции не имеет другого состояния, кроме того, что передается ей в аргументах. И опять же этот код более понятный, так как ни за чем не нужно следить, кроме порядка вызовов функций.

ЛОКАЛИЗАЦИЯ КОНТЕКСТНО ЗАВИСИМОЙ ФУНКЦИОНАЛЬНОСТИ

Откладывание создания экземпляра дает и другие преимущества. Взглядите на эту функцию:

```
my_special_type f2(int a, int b) {
    int const x = perform_f2_related_checks(a, b);
    int const y = perform_boundary_checks(a, x);
    int const z = perform_initialization_checks(b, y);
    return {y, z};
}
```

Она не только создает экземпляр объекта в самом конце, но еще и выигрывает от пропуска операции копирования за счет оптимизации возвращаемого значения.

Можно заметить соответствие кода в этом примере рекомендации «P10. Неизменяемые данные предпочтительнее изменяемых». Здесь `x`, `y` и `z` объявлены как константы. Реализовать эту рекомендацию для встроенных типов тривиально просто, а вот более запутанной инициализации понадобится рассмотреть более сложные соображения. Взгляните на следующий фрагмент кода:

```
int const_var;

if (first_check()) {
    const_var = simple_init();
} else {
    const_var = complex_init();
}
```

Очень хотелось бы сделать переменную `const_var` константным значением, но если объявить ее в условном операторе, то после закрывающей фигурной скобки ее имя выйдет из области видимости. Этую задачу можно было бы решить так:

```
int const var = first_check() ? simple_init() : complex_init();
```

но этот прием плохо масштабируется.

Взгляните также на этот класс:

```
class special_cases {
public:
    special_cases(int, int);
    ...

private:
    my_special_type m_mst;
};
```

Как можно догадаться по приведенной выше функции `f2`, конструирование `my_special_type` предполагает выполнение некоторых довольно специфических условий. Как сконструировать переменную-член `m_mst`? Первое, что приходит на ум:

```
special_cases::special_cases(int a, int b)
    : m_mst(f2(a, b))
{}
```

Но если `f2` больше нигде не используется, то получается, что вы создали специальную функцию исключительно для вызова конструктором. Такое понравится не каждому, кто станет сопровождать этот код, возможно,

это будете вы сами. Выход — использование немедленно вызываемого лямбда-выражения (Immediately Invoked Lambda Expression, IILE). IILE — простая идиома:

```
special_cases::special_cases(int a, int b)
: m_mst([]() { // Передача по значению, без параметров
    int const x = perform_f2_related_checks(a, b);
    int const y = perform_boundary_checks(a, x);
    int const z = perform_initialization_checks(b, y);
    return my_special_type{y, z}; }
)() // Немедленный вызов лямбда-выражения
{}
```

Мы объявили лямбда-выражение и тут же его вызвали. Что касается названия, то, напомним, выбор хороших имен — сложная задача, и здесь оказалось достаточно просто сказать, что надо сделать, преобразовать это в аббревиатуру и жить с этим (см. также RAII). Теперь наша функция инициализации находится в одном месте.

Эту идиому можно применить и к другому примеру:

```
int const var = [](){
    if (first_check()) return simple_init();
    return complex_init();
}();
```

Такие локализованные фрагменты функциональности, используемые для инициализации объектов, часто приводят к созданию временного состояния, надобность в котором исчезает после объявления рассматриваемого объекта. Поскольку временное состояние является зависимостью объекта, оно должно объявляться в той же области видимости. Заключение его в лямбда-выражение создает локальную область, способную экспортировать значение.

УСТРАНЕНИЕ СОСТОЯНИЯ

Отложенное объявление применимо даже к контейнерам. Рассмотрим следующую функцию:

```
void accumulate_mst(std::vector<my_special_type>& vec_mst,
                     std::vector<int> const& source) {
    auto source_it = source.begin();
    while (source_it != source.end()) {
        auto s_it = source_it++;
        ...
```

```

    my_special_type mst{*s_it, *s_it};
    vec_mst.push_back(mst);
}
}

```

Внутри цикла `while` создается экземпляр `my_special_type`, который помещается в вектор. Есть возможность полностью отказаться от конструирования экземпляра и втолкнуть в вектор экземпляр правостороннего значения (rvalue):

```

void accumulate_mst(std::vector<my_special_type>& vec_mst,
                     std::vector<int> const& source) {
    auto source_it = source.begin();
    while (source_it != source.end()) {
        auto s_it = source_it++;
        vec_mst.push_back(my_special_type{*s_it, *s_it});
    }
}

```

В неоптимизированной сборке будет создан временный объект и вызван `push_back(my_special_type&&)`, что позволит использовать конструктор перемещения вместо конструктора копирования. Но можно пойти еще дальше и использовать `emplace_back`:

```

void accumulate_mst(std::vector<my_special_type>& vec_mst,
                     std::vector<int> const& source) {
    auto source_it = source.begin();
    while (source_it != source.end()) {
        auto s_it = source_it++;
        vec_mst.emplace_back(*s_it, *s_it);
    }
}

```

Мы устранили некоторое состояние из функции. Теперь у вас может сложиться впечатление, что тем самым мы повысили производительность программы. Однако все представленные функции делают одно и то же, и, скорее всего, компилятор сгенерирует для каждой один и тот же ассемблерный код. В первом примере компилятор видит, что экземпляр `my_special_type` не используется за пределами цикла `while` и может вызывать `push_back(my_special_type&&)` вместо `push_back(my_special_type const&)`. В третьей функции мы еще больше откладываем конструирование объекта, но это всего лишь перемещение пропуска копирования в другое место. Благодаря исключению копирования объект будет сконструирован один раз и сразу в нужном месте. Фактически для `emplace_back` компилятор генерирует более медленный код, чем для `push_back`: это шаблонная функция-член шаблона класса, а не просто функция-член шаблона класса.

Последнее обстоятельство может повлиять на ваше решение о ее возможном использовании.

В этих примерах предполагается, что конструкторы перемещения и копирования установлены и имеют тривиальные реализации по умолчанию. Если конструктор перемещения отсутствует или его выполнение обходится дорого, можно рассмотреть возможность использования `emplace_back`. Но в остальных случаях используйте `push_back` и конструируйте объекты на месте.

ПОДВЕДЕМ ИТОГ

Вы можете спросить: «Коль скоро для всех этих функций генерируется один и тот же код, то какая разница, как их записывать? Почему я должен предпочесть один способ другому?» Ответ прост: эта рекомендация способствует упрощению сопровождения, но не увеличению производительности. Объявление в момент использования улучшает читаемость кода, а отсутствие объявления состояния улучшает читаемость еще больше. Для рассуждений о состоянии требуется запоминать множество мелких деталей, что становится практически невозможно с расширением кодовой базы.

Откладывание объявления до самого последнего момента дает несколько преимуществ. Ненужное состояние можно уменьшить в объеме или даже вообще исключить. Область видимости сводится к минимуму. Объекты не используются до инициализации. Улучшается читаемость. Все это способствует повышению безопасности, а иногда и упрощению сопровождения кода. Думается, что этому золотому правилу нужно следовать, насколько это возможно.

ГЛАВА 5.5

Per.7. При проектировании учитывайте возможность последующей оптимизации

МАКСИМАЛЬНАЯ ЧАСТОТА КАДРОВ

Я зарабатываю на жизнь играми¹. Точнее, я зарабатываю на жизнь, помогая создавать игры, так как работаю в команде из нескольких сотен человек, каждый из которых имеет свою специализацию. Помимо аниматоров, художников, звукоинженеров, тестировщиков и продюсеров в нашей команде работает много инженеров.

Так было не всегда. Раньше, в 1980-х, я создавал игры в одиночку, и игры выживали или умирали в зависимости от поддерживаемой ими частоты кадров. В Великобритании в роли дисплеев для домашних компьютеров использовались семейные телевизоры PAL, включавшиеся в электросеть переменного тока с частотой 50 Гц. Дисплеи с электронно-лучевыми трубками воспроизводили изображение методом чересстрочной развертки в чередующихся кадрах, поэтому разработчики стремились уместить длительность игрового цикла в 40 миллисекунд, чтобы добиться частоты воспроизведения 25 кадров в секунду.

Это была жесткая константа, связанная с законами физики или по крайней мере с частотой тока в бытовой электросети. Я создавал игры для Sinclair ZX Spectrum, работавшего на процессоре Z80. Я знал, сколько микросекунд выполняется каждая инструкция. Благодаря тщательному учету я знал, сколько строк развертки пробежит луч за время выполнения каждого

¹ Экскурс в историю с Гаем Дэвидсоном.

фрагмента (я не решаюсь здесь использовать слово «функция»). Добавившись до 625, я понимал, что мое время истекло.

Казалась невозможной любая мысль о продаже игр на рынке США, где электросети переменного тока имели частоту 60 Гц, а процессор работал с прежней скоростью. На игровой цикл оставалось меньше времени, и мне пришлось бы урезать функции, чтобы добиться более высокой частоты кадров. Одним из преимуществ появления телевидения высокой четкости было постоянство разрешения и частоты кадров: 1920×1080 пикселей и 60 Гц. К сожалению, когда рынок снова разделился между 4KTV и HDTV, этот период стабильности закончился. Остается надеяться, что гонка аппаратных вооружений остановится на 4K.

В разработке видеоигр в 1980-х годах процесс оптимизации означал одно и только одно: более быстрое выполнение в меньшем пространстве памяти. Только для скорости не было никакой оптимизации: из-за ограниченного объема ОЗУ развертывание цикла означало отъем денег у Питера, чтобы заплатить Полу. Точно так же оптимизация, только по объему используемой памяти, означала увеличение времени, необходимого для отображения кадра.

Одной из распространенных уловок было встраивание вручную. Написав и отладив функцию, можно было решить, вызывать ли ее или вставить ее тело в место вызова. Зная, сколько потребуется времени для выполнения каждой инструкции и оперативной памяти для такой подстановки, можно очень точно определить прирост скорости.

Подстановка отличается от развертывания цикла. При развертывании тело цикла дублируется один или несколько раз (последнее практикуется, чтобы уменьшить количество возвратов в начало цикла). Конечно, это влечет дополнительные затраты памяти. А вот замена вызова функции ее телом иногда может даже сэкономить память за счет избавления от необходимости сохранять и восстанавливать регистры до и после вызова функции.

Манипулирование стеком было моим наиболее излюбленным приемом. Когда процессор встречает инструкцию вызова функции, он сохраняет программный счетчик в стеке и записывает в него operand инструкции вызова. Когда процессор встречает инструкцию возврата, он выталкивает содержимое стека в программный счетчик. Такой порядок вещей прекрасно поддерживает идею единой точки входа и единой точки возврата, но ее можно красиво обыграть для создания таблиц переходов. Если известно,

что последовательность функций должна выполняться в определенном порядке, можно поместить адреса этих функций в стек в порядке, обратном желаемому выполнению, а затем выполнить инструкцию возврата. Каждая функция будет выполняться и «возвращаться» к следующей функции в стеке. Этот прием давал существенную экономию памяти, но был чертовски сложен в отладке и полностью не переносим.

Я обнаружил, что оптимизация — самая забавная часть создания игр. Можно было продолжать и продолжать оптимизировать, пока не возникало решение, что «достигнут конец строки». После этого функция объявлялась оптимальной. Оптимизация основывалась на глубоком и детальном знании платформы. Когда я работал с процессором Z80, я знал, что инструкции загрузки регистров имеют коды между 0x40 и 0x7f, что инструкции условного возврата начинаются с кода 0xc, 0xd, 0xe или 0xf и заканчиваются 0x0 или 0x8 и что инструкции xor имеют коды от 0xA8 до 0xAF, в порядке регистров-операндов b, c, d, e, h, l, (hl), a.

РАБОТА ВДАЛЕКЕ ОТ ЖЕЛЕЗА

Ныне все по-другому. C++ очень близок к железу, но это не машинный язык. Компиляторнейтрализует разницу между ними и несет в себе все подробные знания, которые у меня были раньше о процессоре Z80. В настоящее время это касается архитектуры x86-64, которая включает множество процессоров. Я уже не могу приобрести необходимые достаточно глубокие знания такого рода самостоятельно. Это одна из причин моего перехода на C++.

Есть искусство оптимизации, но есть искусство разработки кода, который хорошо поддается оптимизации. Нельзя оптимизировать код, абсолютно негибкий, каковым часто оказывается оптимизированный код. Оптимизируемый код оставляет возможности для оптимизатора, будь то человек или компилятор. Код, который можно оптимизировать, — это код, допускающий возможность что-то изменить. В действительности слова *optimize* («оптимизировать») и *option* («выбор») имеют один и тот же латинский корень *optare* — «выбирать».

Инженеры обычно оптимизируют код двумя способами: улучшая его структуру или повышая производительность. Улучшение структуры кода находит отражение в том, что он становится ясным и понятным, появляется реальная возможность его повторного использования, упрощается

его сопровождение, повышается скорость его доработки или изменения при уточнении требований и условий работы с ним. Улучшение производительности означает предоставление компилятору максимального объема информации о том, что разработчик хочет сделать, и самого широкого набора вариантов и инструментов для фактического роста производительности программы.

На странице с описанием этой рекомендации в Core Guidelines приводится хороший пример из стандарта C, который мы воспроизведем здесь:

```
void qsort(void* base, size_t num, size_t size,
           int(*comp)(const void*, const void*));
```

Эта функция принимает указатель на некоторую область в памяти, количество элементов, которые нужно отсортировать, размер элементов и функцию сравнения.

Начнем с оптимизации структуры этой функции. Первая проблема заключается в том, что, по всей видимости, мы сортируем память, а не элементы. Это неправильный уровень абстракции. Имя `sort`, данное функции, подразумевает сортировку некоторых элементов. Идея сортировки памяти бессмысленна. Кроме того, принимая `void*`, функция теряет информацию, известную в точке вызова: тип объекта.

Мы можем исправить эту оплошность, перейдя на C++ и заменив `void*` параметром шаблона:

```
template <typename T>
void sort(T* base, size_t num, size_t size,
          int(*comp)(const T*, const T*));
```

Эта оптимизация улучшает структуру и помогает обнаружить типичную ошибку во время компиляции, а не во время выполнения: передачу бессмысленной функции сравнения. Указатель `void*` может ссылаться на что угодно, тогда как `T*` должен указывать на экземпляр `T`. Если предполагается отсортировать множество значений с плавающей точкой, а указанная функция сравнения сопоставляет символы, то вызов `sort` может дать неправильный результат.

Теперь рассмотрим два параметра `size_t`. Во-первых, эти параметры — отложенная катастрофа, ожидающая своего часа. Клиент должен позаботиться о том, чтобы передать эти параметры в правильном порядке; он не получит предупреждения от компилятора, если ошибется. Однако параметр `size`

теперь оказывается избыточным. Сортируемый тип является частью сигнатуры функции, а значит, его размер известен во время компиляции. Теперь сигнатура выглядит так:

```
template <typename T>
void sort(T* base, size_t num, int(*comp)(const T*, const T*));
```

Специализация функции типом дает компилятору больше информации. Если общий объем памяти, занимаемый сортируемыми объектами, меньше, например, размера кэша L1, открывается дополнительная возможность оптимизации.

Последний параметр — указатель на функцию. Это функция обратного вызова. Функция `sort` будет вызывать ее, чтобы определить порядок двух объектов. Здесь трудно изыскать возможность оптимизации, но, поскольку `sort` теперь является шаблоном функции, разумно надеяться, что конкретная функция будет создана во время компиляции. Если определение функции обратного вызова видно функции `sort`, его можно встроить в специализацию шаблона функции.

Однако есть решение лучше — сделать функцию обратного вызова параметром шаблона. Взгляните на следующую сигнатуру:

```
template <typename T, typename F>
void sort(T* base, size_t num, F fn);
```

Она накладывает некоторые ограничения на тип `F`. Он должен быть вызываемым, принимать два параметра `T*` и возвращать отрицательное целое значение, если первый аргумент предшествует второму; ноль, если они равны; и положительное целое значение в противном случае. В конце концов, именно это и делал первоначальный указатель на функцию. Однако теперь можно передать лямбда-функцию, а не указатель на функцию, что повышает вероятность встраивания. Также можно передать объект `std::function` или даже указатель на функцию-член.

Мы знаем, что в эту функцию будут передаваться значения, поэтому можем ограничить ее еще больше и передать пару `T const&`, что расширит диапазон возможных функций и избавит от необходимости проверять пустые указатели. Это дополнительная информация для компилятора и дополнительные возможности для оптимизации.

Что дальше? Первые два параметра сейчас выглядят немного подозрительно. Функция `qsort` требует, чтобы сортируемые элементы занимали

непрерывную область памяти. Это дает несколько преимуществ. Во-первых, можно просто передать пару итераторов. Вероятнее всего, начать все равно придется с пары итераторов и применить `std::distance` для вычисления количества элементов. Используя правильный уровень абстракции, необходимо передать диапазон значений для сортировки:

```
template <typename InIt, typename F>
void sort(InIt first, InIt last, F fn);
```

Это более безопасное определение, потому что избавляет от необходимости заниматься какими-либо арифметическими вычислениями. Вам нужно только сообщить функции концы диапазона. Это обеспечивает оптимизацию структуры за счет уменьшения вероятности ошибки, а также оптимизацию компиляции за счет объявления адресов первого и последнего элементов вместо требования к компилятору создать код для их вычисления. Теперь, находясь в дивном новом мире C++20, мы можем еще больше улучшить структуру, добавив предложения `requires`. На самом деле мы можем поступить еще лучше и заменить пару итераторов диапазоном:

```
template <typename R, typename F>
void sort(R&& r, F fn);
```

Далее, используя концепцию `random_access_range`, можно добавить осмысленные ограничения, еще больше упростив структуру кода и предоставив компилятору еще больше информации:

```
template <std::ranges::random_access_range R, typename F>
void sort(R&&, F fn);
```

Вы наверняка уже догадались, что здесь мы проследили эволюцию `sort`, начиная с эпохи, предшествовавшей появлению C++, и заканчивая C++20. Функцию `qsort` все еще можно найти в стандартной библиотеке, но обычно ее используют только в устаревших окружениях. Приведенная выше сигнатура функции не соответствует стандарту; диапазоны работают с проекциями, обсуждение которых выходит за рамки этой главы, но для完整性 картинки мы хотим показать, как выглядит сигнатура одной из ее перегруженных версий:

```
template <std::ranges::random_access_range R,
          typename Comp = std::ranges::less, typename Proj =
          std::identity>
requires std::sortable<std::ranges::iterator_t<R>, Comp, Proj>
constexpr std::ranges::borrowed_iterator_t<R>
sort(R&& r, Comp comp={}, Proj proj={});
```

Эта функция:

- принимает диапазон, функцию сравнения и проекцию;
- требует отсортировать диапазон экземпляров указанного типа;
- является константным выражением;
- возвращает итератор, указывающий на конечный элемент.

Кроме того, эта функция дает компилятору гораздо больше информации, чем `qsort`, и обеспечивает более четкое место вызова.

Однако, как можете видеть сами, размер объявлений значительно увеличился. К сожалению, заметно увеличилось также время компиляции. Вместо того чтобы, как в случае с `qsort`, просто вставить вызов функции в нужное место, компилятор сначала определяет параметры шаблона, проверяет ограничения, создает экземпляр функции, а затем оптимизирует его и оценивает возможность встраивания. Время выполнения кода в данном случае сокращается за счет увеличения времени компиляции.

Оптимизация требует времени. Такие затраты — разумное вложение. Будьте готовы изменить структуру кода, чтобы улучшить производительность. Но не торопитесь вооружаться виртуальным секундомером и прочесывать каждую функцию. Давайте сначала определимся, нужны ли вам эти вложения.

ОПТИМИЗАЦИЯ ЧЕРЕЗ АБСТРАКЦИЮ

Вспомните название этой главы: «При проектировании учитывайте возможность последующей оптимизации». Набор только что описанных шагов можно с тем же успехом включить в главу об абстракции. Мы не только улучшили производительность функции, но и абстрагировали ее, эти два явления часто наблюдаются вместе. Но необходимо задать вопрос: не была ли эта оптимизация преждевременной?

В 1990-е часто звучало предупреждение о том, что «преждевременная оптимизация — корень всех зол». Обычно его произносили, когда кто-то тратил время на оптимизацию функции вручную до того, как требования устоятся. В разных источниках эта фраза приписывается Дональду Кнуту (Donald Knuth) или Тони Хоару (Tony Hoare). Впрочем, любой из них вполне мог процитировать другого. Оба являются замечательными учеными и, без преувеличения, могут быть сравнимы с мифическими гигантами, на

плечах которых стоим все мы. Ховар изобрел быструю сортировку. Кнут изобрел TeX. Мы должны прислушиваться к тому, что они говорят, но при этом также хорошо понимать, что именно они говорят, и уметь отличать случаи, когда их фразы используются не к месту.

В статье *Structured Programming with go to Statements*, опубликованной в журнале *Computing Surveys*, том 6, номер 4, за декабрь 1974 года, Дональд Кнут писал: «Нет никаких сомнений в том, что Грааль эффективности приводит к злоупотреблениям. Программисты тратят массу времени, думая или беспокоясь о скорости маловажных частей программ, и эти попытки повысить эффективность на самом деле оказывают большое отрицательное влияние на отладку и сопровождение. Мы должны забывать о плохой эффективности, скажем, в 97 % случаев: преждевременная оптимизация — корень всех зол.

И напротив, мы должны уделить внимание оставшимся критически важным 3 %. Хорошего программиста такие рассуждения не убаюкают, у него достанет мудрости внимательно исследовать критический код, но только после того, как этот код будет выявлен. Часто не следует спешить делать априорные суждения о том, какие части программы действительно важны, потому что обобщенный опыт программистов, использующих инструменты измерения, показывает, что их интуитивные догадки нередко оказываются ложными.

Поработав с такими инструментами в течение семи лет, я пришел к убеждению, что все компиляторы, написанные с этого момента, должны разрабатываться так, чтобы дать всем программистам информацию о том, какие части их программ обходятся дороже всего с точки зрения времени выполнения. Эта информация должна предоставляться автоматически, если только ее вывод не окажется специально отключен».

Распространенное неправильное употребление обусловлено принципом Парето, согласно которому 80 % времени выполнения приходится на 20 % кода. Исходя из этого, время для оптимизации наступает в конце разработки, когда можно определить эти 20 %. Однако если эти 20 % рассеяны по всей кодовой базе, то безопасная модификация становится практически невозможной. Кроме того, как предполагает Кнут, очень сложно, рассматривая исходный код, выявлять критические точки производительности. Это лишает инженера мотивации оптимизировать разделы кода, потому что они как таковые не могут с уверенностью сообщить, повлияло ли на их производительность время, потраченное на доработку.

Оптимизация должна происходить вместе с рефакторингом по мере стабилизации требований. Оптимизация всей программы — дурацкая затея.

Наибольшее влияние на производительность оказывает выбор правильного алгоритма, а уж создание кода, подходящего для реализации алгоритма, позволит и оптимизировать его. В прежние времена можно было просто подождать, пока будет создан и внедрен более быстрый процессор, но те времена давно миновали, сейчас на это полагаться не стоит.

Подумайте, что означает цифра 3 %: это три строки кода в каждой сотне. Насколько длинны ваши функции? Высока вероятность, что с производительностью коротких пятистрочных функций все в порядке, а вот более длинные функции могут выиграть от тщательного изучения. Лучшие оптимизации — те, что улучшают ясность вашего замысла и увеличивают объем информации, предоставляемой компилятору, например, когда циклы, написанные вручную, заменяются стандартными алгоритмами.

Вернемся к примеру с сортировкой. Одно из различий между `qsort` и `std::ranges::sort` заключается в возвращаемом типе. Вместо `void`, как это делает `qsort`, функция `std::ranges::sort` возвращает итератор, равнозначный концу диапазона. Особой нужды в этом нет, но может пригодиться в некоторых случаях. Один из способов разработки с учетом возможности оптимизации состоит в том, чтобы предоставить семейство функций, реализующих один и тот же фундаментальный алгоритм, но возвращающих разные объемы информации.

Стандартные алгоритмы поддерживают это. Функция `std::is_sorted` принимает диапазон и функцию сравнения и возвращает значение `bool`, сообщающее, отсортированы ли элементы. `std::is_sorted_until` возвращает итератор, указывающий на первый элемент в диапазоне, следующий не по порядку. `std::is_sorted` можно реализовать через вызов `std::is_sorted_until` с проверкой совпадения возвращаемого значения с концом диапазона.

Точно так же `std::mismatch` сравнивает два диапазона, выполняя поиск первой пары элементов, не удовлетворяющей предикату. `std::equal` является специализированной версией этой функции: роль предиката играет оператор равенства, и если `std::mismatch` возвращает конец диапазона, то указанные диапазоны равны.

В этих примерах мы продемонстрировали пары функций, реализующих одну идею, но работающих на разных уровнях абстракции. Вы можете выбрать правильную функцию для уровня, на котором работаете, и тем самым дать компилятору не больше и не меньше того, что ему необходимо знать о поставленной задаче, чтобы сгенерировать оптимальный код.

ПОДВЕДЕМ ИТОГ

Оптимизация — это то, что лучше всего переложить на компилятор. У вас может быть свое представление о том, какой код будет генерирован на основе вашего исходного кода, но фактический код генерирует компилятор. Сообщите компилятору о своих намерениях как можно более полную и точную информацию, чтобы получить максимальную отдачу от него.

Сосредоточьтесь на разработке интерфейсов с поддержкой возможности композиции. Разрабатывая функцию, спросите себя, можно ли составляющие ее части выразить как отдельные функции, которые могут иметь самостоятельное применение. Оптимизация вручную не единственный способ улучшить код, и, скорее всего, это преждевременная оптимизация. Тщательная композиция функций и правильное обобщение обеспечат гораздо лучшие возможности для оптимизации.

Все имеет свою цену. Вы можете знать, сколько наносекунд занимает выполнение процессорной инструкции, сколько микросекунд нужно для переключения контекста потока, сколько миллисекунд требуется для отправки буфера размером 1 Кбайт через полмира и т. д. Сравнение порядков величин даст вам больше, чем сравнение самих значений величин. Оставьте внутреннюю работу компилятору.

ГЛАВА 5.6

Е.6. Используйте идиому RAII для предотвращения утечек памяти

ДЕТЕРМИНИРОВАННОЕ УНИЧТОЖЕНИЕ

Выше в этой книге мы уже увлеченно рассуждали о детерминированном характере уничтожения объектов, но хотим повторить еще раз: это уникальная величайшая особенность C++. К сожалению, в языке есть небольшая проблема, связанная с классами хранения объектов.

Существует четыре класса хранения. Объекты со статическим классом хранения создаются до вызова `main()` и уничтожаются после возврата из `main()`. Объект с автоматическим классом хранения создается сразу после объявления и уничтожается, когда его имя выходит за пределы области видимости. Объект с локальным для потока классом хранения создается в момент запуска потока выполнения и уничтожается по завершении потока.

Динамический класс хранения подразумевает создание и уничтожение объекта по явному требованию пользователя. Динамические объекты создаются с помощью оператора `new` и уничтожаются с помощью оператора `delete`. Эти операторы являются частью исходного кода, написанного пользователем. Однако известно, что код, написанный пользователем, уязвим для ошибок. Существует два типа ошибок, связанных с управлением объектами: попытка использовать объект после уничтожения и полная потеря существующего объекта (например, потому, что вы забыли вызвать оператор `delete`). Сейчас нас интересует последняя ошибка. Потеря объекта называется утечкой памяти.

Вот простейшая возможная утечка:

```
#include <vector>
int main() {
    new std::vector<int>;
}
```

Это вполне допустимый код, он опробован в Compiler Explorer с несколькими компиляторами, и все они компилируют его без вывода предупреждений. При выполнении оператор `new` запрашивает объем памяти, необходимый для экземпляра `std::vector<int>`, обычно три слова. Система выделяет память и возвращает ее оператору `new`, который затем вызывает конструктор по умолчанию `std::vector<int>`, чтобы заполнить эти три слова. Затем `main()` завершается.

Объект не связан с именем. В этом не было необходимости. Оператор `new` сделал свою работу в соответствии с инструкциями. Но, не привязав результат оператора `new` к имени, его нельзя будет удалить, поэтому сразу возникла утечка. Память не была освобождена автоматически после выхода из функции `main()`, хотя, скорее всего, по завершении процесса операционная система выполнит соответствующую очистку. Она не будет вызывать никаких деструкторов, но вернет себе память.

Вот еще одна простейшая утечка:

```
#include <vector>
int main() {
    auto vi = new std::vector<int>;
}
```

На этот раз мы привязали результат оператора `new` к имени. К сожалению, имя покинуло область видимости до того, как мы удалили объект. В данном случае, когда имя вышло за пределы области видимости, автоматически уничтожился не объект `std::vector<int>`, а указатель на него. Переменная `vi` имеет тип `std::vector<int>*`, и когда имя объекта, на который она указывает, выходит за пределы области видимости, то уничтожается только сам указатель, а объект, на который он ссылался, остается в памяти.

До появления C++11 с его интеллектуальными указателями подобные проблемы были чрезвычайно распространены. Теперь мы можем написать:

```
#include <vector>
#include <memory>

int main() {
    auto vi = std::unique_ptr<std::vector<int>>(new std::vector<int>);
}
```

Как громоздко! Однако после появления `std::make_unique` в C++14 можно написать:

```
#include <vector>
#include <memory>

int main() {
    auto vi = std::make_unique<std::vector<int>>();
}
```

Так намного яснее, и мы надеемся, что вы согласитесь с этим. Теперь `vi` уже не указатель типа `std::vector<int>*`, а объект типа `std::unique_ptr<std::vector<int>>`.

Жизненный цикл этой версии переменной `vi` несколько изменился. Теперь он инициализируется не только адресом памяти, выделенной при создании, но также инструкциями, описывающими порядок уничтожения объекта. То есть, когда имя `vi` выходит из области видимости, вызывается деструктор объекта, с которым оно связано, а тот, в свою очередь, уничтожает объект `std::vector<int>`.

Эта идиома, в которой создание объекта также включает сведения о том, как его уничтожить, известна как идиома получения ресурсов при инициализации (Resource Acquisition Is Initialization, RAII). Жизненный цикл выделенной памяти привязан к жизненному циклу объекта. Сама фраза и аббревиатура были придуманы Бьерном Страуструпом и впервые появились в его книге *The C++ Programming Language*¹.

Концепция RAII дает чрезвычайно полезное преимущество. Она применима не только к памяти, но и к любому ресурсу, жизненный цикл которого требует явного управления. Оставшуюся часть этой главы мы посвятим изучению примера.

Следует отметить, что примеры выше создавались с целью продемонстрировать утечку памяти. Их можно исправить, создав вектор в стеке, то есть с автоматическим классом хранения.

Если есть такая возможность, всегда отдавайте предпочтение автоматическому классу хранения, а не динамическому. Используйте динамические объекты, только если они должны сохраняться после выхода из текущей области видимости. Если вы новичок в C++, то это пожелание может вас удивить и заинтересовать.

¹ Страуструп Б. Язык программирования C++.

Возможно, вы никогда не вызывали оператор `new` и всегда пользовались только интеллектуальными указателями для создания больших объектов, которые дешевле передавать по указателю. Возможно, вам когда-то уже говорили, что, отказываясь от простых указателей, вы избегаете утечек памяти. Однако утечка памяти не единственный вид утечки.

УТЕЧКА ФАЙЛОВ

Программистам, пишущим программы для Windows, хорошо знакома функция `CreateFile`. Она создает или открывает файл или устройство ввода/вывода и возвращает дескриптор соответствующего объекта. Дескриптор имеет тип `HANDLE` — псевдоним для `void*`. Дескриптор используется в вызовах всех функций, выполняющих операции с файлом: `ReadFile`, `WriteFile`, `SetFilePointer` и т. д. Когда объект становится ненужным, вызывается функция `CloseHandle`, которая освобождает ресурс и возвращает его операционной системе.

То же верно в отношении функций для работы с файлами из стандартной библиотеки. Те, кто решил не использовать библиотеку `iostreams`, обычно вызывают функцию `std::fopen`, которая создает либо открывает файл или устройство ввода/вывода и возвращает указатель на определяемый реализацией тип с именем `FILE`. Этот указатель используется в вызовах всех функций, выполняющих операции с файлом: `std::fread`, `std::fwrite`, `std::fseek` и т. д. Когда объект становится ненужным, вызывается функция `std::fclose`, которая возвращает ресурс операционной системе.

Между этими наборами операций можно заметить определенное сходство. Функция `std::fread` принимает указатель на `FILE` в последнем аргументе, а `ReadFile` принимает указатель на `HANDLE` в первом аргументе. Функция `std::fread` читает указанное количество объектов заданного размера, а `ReadFile` читает указанное количество байтов, но принцип тот же: для выполнения операций с файлом вы используете дескриптор, полученный от операционной системы.

Дескрипторы могут утекать, как и память. Вот пример простейшей возможной утечки:

```
#include <cstdio>

int main() {
    std::fopen("output.txt", "r");
}
```

`std::fopen` возвращает указатель `FILE*`, но программа не связывает его ни с каким именем, и поэтому он утекает. Можно повторить и второй пример:

```
#include <cstdio>

int main() {
    auto file = std::fopen("output.txt", "r");
}
```

Здесь тоже имеет место утечка дескриптора `FILE`. В этом примере дескриптор привязан к имени, но отсутствует вызов функции `std::fclose`, который вернул бы ресурс операционной системе.

К счастью, решить эту проблему поможет стандартная библиотека C++ — `iostreams`. Она предлагает на выбор несколько объектов с управляемыми жизненными циклами. Точно так же, как `std::unique_ptr` освобождает ресурс памяти, когда тот выходит за пределы области видимости, объекты из библиотеки `iostream` освобождают дескрипторы. Например:

```
#include <fstream>

int main()
{
    auto file = std::fstream{ "output.txt" };
}
```

Чтобы открыть файл, его имя передается конструктору объекта `std::fstream`. Он откроет файл, позволит вам вызывать функции-члены, такие как `read`, `write`, `seekp` и `seekg`, и закроет файл при вызове деструктора.

Библиотека `iostreams` нравится не всем. Она разработана с использованием абстрактных базовых классов, что снижает ее производительность. Эта библиотека — продукт своего времени, то есть начала 1990-х годов. С тех пор разработчики узнали множество новых правил о проектировании библиотек C++, например, открыли для себя значение композиции, и если бы они начали заново, то, думается, выбрали бы другой подход для реализации `iostreams`. Но это по-прежнему очень хорошая библиотека, которая выполняет то, что обещает, хотя многие программисты испытывают искушение использовать свои собственные подходы и писать свои библиотеки для работы с файлами.

Есть более простые способы решить проблему утечки файлов. Один из них заключается в создании объекта наподобие `std::unique_ptr`, который вместо указателя на память содержит файл. Например:

```
#include <cstdio>

class FILE_holder {
public:
    FILE_holder(std::FILE* f) : m_f(f) {}
    ~FILE_holder() { std::fclose(m_f); }
    operator std::FILE*() { return m_f; }

private:
    std::FILE* m_f;
};

int main()
{
    auto file = FILE_holder(std::fopen("output.txt", "r"));
}
```

Ниаких утечек. Конечно, есть еще одна проблема: удаление от этого объекта может привести к преждевременному закрытию файла. На самом деле нам нужно нечто похожее на `std::unique_ptr`, но предназначенное для объектов, созданных с помощью `std::fopen`, а не с помощью оператора `new`.

К счастью, комитет подумал и об этом. `std::unique_ptr` — это шаблон класса не с одним, а с двумя параметрами. Первый параметр — тип хранимого объекта, а второй параметр — средство удаления. Второй параметр принимает значение по умолчанию `std::default_delete` — очень простой объект с конструктором и оператором круглых скобок. Простейшая его реализация могла бы выглядеть так:

```
template<class T>
struct default_delete {
    constexpr default_delete() noexcept = default;
    template<class U>
    default_delete(const default_delete<U>&) noexcept {}
    void operator()(T* p) const { delete p; }
};
```

Вместо того чтобы писать свой инструмент удаления и использовать его при создании экземпляров `std::unique_ptr`, можно специализировать шаблон для `std::FILE`. Это просто, как показано ниже:

```
#include <memory>
#include <cstdio>

template <>
struct std::default_delete<std::FILE> {
```

```
void operator()(std::FILE* f) { std::fclose(f); }

int main()
{
    auto file = std::unique_ptr<std::FILE>(std::fopen("output.txt", "r"));
}
```

Специализация заключается в замене оператора круглых скобок, вызывающего оператор `delete`, вызовом `std::fclose`. Когда `file` покидает область видимости, объект `std::unique_ptr`, содержащий `std::FILE*`, уничтожается и попутно закрывает объект `std::FILE*`.

ПОЧЕМУ ЭТО ТАК ВАЖНО

В идеале все классы, связанные с какими-либо ресурсами, должны иметь конструкторы и деструкторы, управляющие жизненным циклом этих ресурсов. Как мы видели на примере `std::FILE*`, у нас есть запасной выход для несоответствующих объектов, но что делать, если наш ресурс не отображается как указатель?

Возможно, вам кажется, что мы прилагаем слишком много усилий, чтобы прибрать за собой память, тогда как операционная система готова сделать все это за нас. Когда процесс завершается, все дескрипторы, связанные с ним, закрываются, вся память освобождается и все ресурсы оказываются готовы к повторному использованию. Так зачем беспокоиться об утечке ресурсов, если окружение все равно позаботится об этом?

Тому есть несколько причин. Во-первых, если ресурсы утекают достаточно быстро, наступит момент, когда операционная система отклонит очередной ваш запрос, предупредив, что не осталось ничего из того, что вы запрашиваете. Это особенно вероятно, если программа выполняется продолжительное время и предназначена для работы в течение всего времени безотказной работы компьютера, на котором она развернута.

Во-вторых, это хорошая привычка. Если вы решите, что прибирать за собой не обязательно, это будет означать, что теперь каждый раз, создавая что-то, что может утекать, вы будете тратить время на решение, стоит ли тратить время на разработку кода, предотвращающего утечку. Возьмите за правило всегда прибирать за собой.

В-третьих, для файлов: в некоторых операционных системах, пока работающее приложение удерживает файл открытым, пользователь не сможет изменить, переместить или удалить этот файл. Это может вызывать раздражение и иногда даже заставлять пользователей перезагружать свои компьютеры, чтобы удалить ненужный файл.

Наконец, не всегда безопасно полагаться на то, что операционная система выполнит очистку. Если вы используете устаревшую стороннюю библиотеку, утечка ресурсов может иметь долгосрочные последствия. Взгляните на этот фрагмент:

```
int open_database(const char*);  
void close_database(int);  
  
int main()  
{  
    auto db = open_database("//network/customer.db");  
}
```

Где-то далеко от вашего компьютера существует база данных. Этот код открывает соединение с ней и забывает его закрыть. Операционная система ничего не знает о том, как очистить этот ресурс после утечки. Если повезет, то сервер базы данных, будучи надежным программным обеспечением, сам закроет неактивное соединение по прошествии некоторого времени. Но это небезопасное предположение.

Однако мы не можем специализировать `std::default_delete`, потому что `open_database` не возвращает указатель. У вас может возникнуть соблазн использовать `reinterpret_cast`, чтобы превратить `int` в указатель. Но тогда готовьтесь выдержать суровый осуждающий взгляд того, кто будет проверять ваш код, потому что вы явно лжете компилятору. Правильное решение — создать прокси-объект, например, так:

```
#include <memory>  
  
int open_database(const char*);  
void close_database(int);  
  
struct DATABASE_PROXY {  
    DATABASE_PROXY(int db_) : db(db_) {}  
    operator int() { return db; }  
    int db;  
};
```

```

template <>
struct std::default_delete<DATABASE_PROXY> {
    void operator()(DATABASE_PROXY* p) { close_database(*p); }
};

int main()
{
    auto db = std::unique_ptr<DATABASE_PROXY>
        (new DATABASE_PROXY(open_database("//network/customer.db")));
}

```

Класс DATABASE_PROXY обертывает возвращаемое значение, что позволяет разместить копию в свободном хранилище и передать ее конструктору std::unique_ptr. Этот прием можно также использовать с объектами больше, чем int. Правда, тут можно надеяться, что если структура возвращается из функции, то соответствующее управление ресурсами будет осуществляться именно в этой функции.

ВСЕ ЭТО ВЫГЛЯДИТ ЧЕРЕСЧУР СЛОЖНЫМ: БУДУЩИЕ ВОЗМОЖНОСТИ

Создание структуры только для специализации std::default_delete кажется непропорционально большим объемом работы. Однако в действительности это прием из категории «работа с устаревшим кодом». Мы, сообщество программистов, многому научились за последние 40 лет, и большую часть этих знаний воплотили в положения стандарта языка. Приспособливать код, который считался хорошо написанным в прошлом, но не выиграл от появления последующих идиом и практик, всегда будет дорого.

Например, приведение типов было совершенно нормальным и приемлемым способом работы с конфликтующими типами при написании кода на С. Но C++ укрепил систему типов теоретически и практически, добавив ключевые слова приведения, такие как static_cast и reinterpret_cast, довольно уродливые и указывающие на тот факт, что вы пытаетесь подорвать важные устои языка, а именно безопасность типов. Современный способ моделирования RAII заключается в правильном использовании конструктора и деструктора. Все ресурсы должны приобретаться в конструкторах, освобождаться в деструкторах и правильно управляться операторами присваивания. Ресурсы должны абстрагироваться в собственных классах

с собственными специальными функциями, чтобы избавить клиентские классы от бремени управления ими. Это вплотную подводит к правилу пяти/нуля.

Но давайте заглянем в будущее. В *C++ Extensions for Library Fundamentals, Version 3*¹, в разделе [scopeguard] представлена еще более явная поддержка такой позиции. Там описывается заголовок с именем `<experimental/scope>` (имя изменится, если эта поддержка будет принята в стандарт), который определяет четыре класса:

```
template <class EF> class scope_exit;
template <class EF> class scope_fail;
template <class EF> class scope_success;
template <class R, class D> class unique_resource;
```

Первые три класса обертывают объект функции `EF` и вызывают его при выходе из области видимости, а `std::experimental::unique_resource` — это универсальная оболочка RAII для дескриптора ресурса `R`, которая владеет ресурсом, управляет им и удаляет с помощью средства удаления `D`, когда `std::experimental::unique_resource` уничтожается.

Первые три класса удобно использовать для реализации идиомы RAII в рамках одной области видимости. Выход из области видимости может произойти множеством способов, особенно если функция слишком длинная. И эти классы гарантируют освобождение ресурса, независимо от выхода из области видимости. Если вам понадобится различать выход по исключению и по успешному завершению, то вы сможете реализовать это с помощью `std::experimental::scope_fail` и `std::experimental::scope_success`. Например:

```
void grow(vector<int>& v) {
    std::experimental::scope_success guard([]{
        std::cout << "Good!" << std::endl; });
    v.resize(1024);
}
```

Эта функция имеет два выхода: либо успешно выполнит `v.resize(1024)`, либо сгенерирует исключение. Объект `std::experimental::scope_success` выведет сообщение в стандартный вывод только в случае успешного изменения размера.

¹ www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4873.html

`std::experimental::unique_resource`¹ очень похож на `std::unique_ptr`, но, в отличие от `std::unique_ptr`, не требует, чтобы ресурс был указателем.

Вернемся к примеру `std::fopen`:

```
#include <experimental/scope>
#include <cstdio>

int main()
{
    using std::experimental::unique_resource;
    auto file = unique_resource(
        std::fopen("output.txt", "r"),
        [] (auto fp){ std::fclose(fp); });
}
```

Здесь есть одна проблема: что будет, если вызов `std::fopen` завершится ошибкой? Нужен какой-то способ сообщить вызывающему коду об ошибке `std::fopen` или любого ресурса, который обертывается таким способом.

Существует также аналог шаблона функции `std::make_unique` с довольно замысловатой сигнатурой:

```
template <class R, class D, class S=decay_t<R>>
std::experimental::unique_resource<decay_t<R>, decay_t<D>>
    std::experimental::make_unique_resource_checked
        (R&& resource, const S& invalid, D&& d)
    noexcept(std::is_nothrow_constructible_v<decay_t<R>, R> &&
             std::is_nothrow_constructible_v<decay_t<D>, D>);
```

Этот шаблон функции принимает тип ресурса, значение, недопустимое для ресурса этого типа, и объект функции удаления; и возвращает объект `std::experimental::unique_resource`. Если было получено недопустимое значение, то объект функции удаления не будет вызываться.

Вот как можно переписать пример с `std::fopen`:

```
#include <experimental/scope>
#include <cstdio>

int main()
{
```

¹ Пространство имен `std::experimental` используется для хранения экспериментальных функций, одобренных Комитетом по стандартам C++. Сущности, начинающие свою жизнь здесь, могут оказаться в пространстве имен `std`, если наберут достаточное количество положительных отзывов.

```
auto file = std::experimental::make_unique_resource_checked(
    std::fopen("potentially_nonexistent_file.txt", "r"),
    nullptr,
    [](auto fptr){ std::fclose(fptr); });
}
```

Здесь мы вызываем `std::experimental::make_unique_resource_checked` с результатом, возвращаемым `std::fopen`, и указываем, что в случае успеха должна быть вызвана `std::fclose`. Если значение, возвращаемое функцией `std::fopen`, равно `nullptr`, то `std::fclose` не будет вызываться.

ГДЕ ВСЕ ЭТО ПОЛУЧИТЬ

Все это — очень полезные инструменты. К сожалению, они могут не предоставляться вашим поставщиком.

Однако это не мешает вам реализовать необходимые инструменты самостоятельно. Подробное описание можно найти в технической спецификации, ссылка на которую приводилась в сноске выше. Найдите тег `[scopeguard]`, известный как стабильный индекс. Там вы отыщете полное описание, как должны работать эти четыре класса и функция, не являющаяся членом. На чтение у вас уйдет не более 15 минут, а сама реализация займет меньше времени, чем вы думаете.

Самостоятельная реализация имеет три преимущества. Самое главное из них: вы сможете сразу же начать использовать объекты, а если они будут приняты в стандарт, что весьма вероятно, то вам потребуется внести в свой исходный код только минимальные изменения. Второе преимущество: вы начнете понемногу узнавать, как определяется стандарт и как реализуются библиотечные функции. Третье преимущество: обнаружив какие-либо ошибки или неясности в технической спецификации, вы сможете сообщить о них редакторам и дать им возможность внести исправления до того, как предложение будет принято в стандарт. Когда что-то попадает в стандарт, исправить это что-то становится довольно сложно. Поэтому предпочтительнее выявлять все ошибки заранее, до развертывания.

Наконец, если вы сочтете это дополнение к языку полезным, то обязательно сообщите об этом комитету и попросите их ускорить его включение в стандарт. И напротив, если вы считаете, что дополнение слишком запутанное, чрезмерно сложное, ненужное или не заслуживает места в стандарте по другим причинам, то тоже сообщите об этом комитету,

подробно изложив свои аргументы. В комитет входят представители многих стран и компаний, которые добровольно работают над улучшением стандарта C++. Они делают то, о чем их просит остальной мир, обсуждая разумность, желательность, осуществимость, и, достигнув согласия, предоставляют стандарт C++, отвечающий потребностям сообщества. Имейте в виду, что прогресс может быть медленным: получение согласия от более чем сотни человек из десятков компаний, отраслей и стран требует времени.

Важно подчеркнуть добровольный характер работы комитета. Здесь не требуется ни специального членства, ни частного приглашения, ни тайных договоренностей: участие в обсуждениях достигается простым присутствием на собраниях комитета и содействием процессу. Кодекс поведения регулирует все действия и обеспечивает открытость и прозрачность обсуждений и принятия решений.

Есть несколько способов принять участие в этом. Что касается авторов, то Дж. Гай Дэвидсон, житель Великобритании, связывался с Британским институтом стандартов British Standards Institute и запрашивал информацию о том, как присоединиться к группе BSI C++. В Канаде, где живет Кейт Грегори, можно связаться с Канадским советом по стандартам Standards Council of Canada. В США — с INCITS. В каждой стране есть своя национальная организация по разработке стандартов, а также свои квоты для платного и бесплатного участия в их работе. Вы можете связаться с институтом стандартов в своей стране и навести справки. Даже если вы не хотите участвовать в развитии языка в полной мере, у вас тем не менее должна быть возможность довести свои взгляды до сведения вашего национального органа.

Не все страны мира представлены в комитете C++, но официальное участие новых стран всегда приветствуется, оно может быть инициировано и предпринято любым, кто готов взаимодействовать со своим национальным органом по стандартизации.

Иногда даже один человек может создать национальный орган. В 2017 году Хана Душикова (Hana Dusíková) посетила CppCon и выступила с ярким докладом о разработанном ею синтаксическом анализаторе регулярных выражений времени компиляции. Она привлекла внимание нескольких постоянных членов комитета, сформировала и созвала чешский национальный орган, а теперь возглавляет исследовательскую группу Study Group 7, Reflection.

Узнать больше о процессе стандартизации можно, посетив <http://isocpp.org/std>. Там вы найдете информацию о том, как связаться с национальным органом, как участвовать в разработке стандартов и заседаниях комитетов, как сообщать о дефектах и как подавать предложения. Там же можно найти документы, описывающие порядок действий. Отыскать нужную страницу можно на сайте комитета ISO по языкам программирования: <https://www.iso.org/committee/45202.html>. Перейдите по ссылке *Participating members*, и вы увидите, какие страны участвуют в процессе стандартизации, названия национальных организаций по разработке стандартов для каждой страны и их статус участия. Что касается статуса: Р-члены (члены-участники) имеют право голоса, а О-члены (наблюдатели) – нет. Если вашей страны нет в списке, то либо в вашей стране нет своей национальной организации по разработке стандартов, либо она не присоединилась к Комитету по стандартам ISO. Теперь-то вы будете знать, сколько работы вас ждет впереди.

Взаимодействие с комитетом и участие в формировании стандарта поможет C++ оставаться языком, к которому хочется обращаться для решения задач разработки программного обеспечения.

Заключение

Когда мы с Кейт приступили к работе над этой книгой, я не знал, какие открытия сделаю в процессе. Часть удовольствия от писательского труда заключается в открытии чего-то нового, прежде вам неизвестного. На моей основной работе в Creative Assembly я часто пишу короткие статьи об использовании конструкций языка и библиотечных функций, при этом процесс разъяснения чего-то в письменной форме помогает мне многое прояснить для себя самого.

Написание 90 000 слов данной книги вместо привычной повседневной тысячи значительно увеличило этот опыт. Главное, что я вынес из этой книги, — абстракции имеют первостепенное значение, и очень важно работать на правильном уровне абстракции. Многие рекомендации можно рассматривать как советы по абстрагированию. В конце главы 5.3 я перечислил некоторые из них.

Еще одна часть моей повседневной работы — проведение собеседований с недавними выпускниками университетов, желающими присоединиться к нашей инициативе выпускников-программистов. Начиная изучать C++, студенты обычно сосредотачиваются на разработке функций, которые, условно говоря, читают и записывают состояние в объекты в программе. Начинающие разработчики относятся к классам как к сосудам для хранения состояния, а не как к абстракциям, сформированным из инвариантов.

Я очень надеюсь, что, прочитав эту книгу, вы в большей степени сосредоточитесь на идее программы как набора небольших абстракций, моделирующих решаемую задачу, и в меньшей — на идее программы как последовательности действий по сценарию. Для C++ характерно сочетание средств абстракции с нулевыми издержками с возможностью достигнуть производительности, сопоставимой с производительностью «голого железа». Если вы будете пользоваться только преимуществами высокой производительности и игнорируете возможности абстрагирования, то ваш код будет трудно поддерживать и использовать в других проектах.

Я также надеюсь, что вас заинтересовали остальные рекомендации в Core Guidelines. По мере развития языка должны развиваться и рекомендации, и авторы приветствуют любые предложения по улучшению, снабженные доказательствами. Точно так же я надеюсь, что нам удалось сподвигнуть вас более внимательно следить за разработкой стандарта C++ и, возможно, даже принять участие в разработке языка или стандартной библиотеки.

С созданием C++ Foundation сообщество C++ заметно оживилось за последнее десятилетие, кульминацией чего стало появление C++20, крупнейшего обновления языка в его истории. Этого бы не произошло без постоянного роста сообщества, роста, который находит отражение в проведении конференций и встреч и в создании блогов. Я люблю этот язык и получаю огромное удовольствие от решения задач на нем. Поэтому я очень надеюсь, что вы расширите свое участие в жизни этого сообщества и поделитесь со всеми нами тем, что узнали, потому что именно так сообщества развиваются и укрепляются.

Всем пока!

Bauu Гай

Послесловие

«Красивый C++» — это не только броское название, но и моя личная цель и то, что я больше всего ценю (и на что надеюсь) в эволюции C++. Несмотря на сложность C++, разработка программ на этом языке в современном стиле дает гораздо более чистый — и да, более красивый — код, чем разработка в старом стиле C++98 или «C с классами».

Но что значит писать на C++ в современном стиле? Закрепление ответа на этот вопрос в едином авторитетном месте было одной из основных причин, почему мы с Бьерном Страуструпом разработали *C++ Core Guidelines*. В этом Руководстве в числе прочего описывается соответствие требованиям, мы знакомили вас с инструментами статического анализа, которые и помогают разработчикам выработать и сохранить «современный стиль». Руководство организовано как энциклопедия или справочник, который можно использовать для поиска ответов на многие вопросы. Однако из хорошей энциклопедии получается ужасный учебник или роман, про который нельзя сказать, что он читается на одном дыхании.

Книгу «Красивый C++» отличает от справочного пособия наличие объяснений и представление ключевых принципов в форме рассказа об истории современного C++. Кейт и Гай создали восхитительное легко читаемое повествование, в котором каждая глава развивает сюжет все дальше и дальше, отправляя нас, читателей, в приятное путешествие за новыми знаниями. Теперь, прочитав эту книгу, вы, вероятно, так же, как и я, оцените, как мастерски авторам удалось разнообразить сухой набор рекомендаций своим глубоким опытом и знаниями. В книге доступно и понятно показано, что многие возможности, появляющиеся по мере развития C++, облегчают чтение, создание и сопровождение кода на C++, потому что позволяют нам напрямую выражать наши намерения средствами обновленного языка. В результате получается красивый код, легко читаемый, простой в сопровождении, надежный и профессиональный.

Как неоднократно говорил Бьерн Страуструп, внутри C++ находится маленький элегантный язык, изо всех сил пытающийся выбраться наружу. Современные подходы к использованию C++ — важный шаг на пути к овладению этим маленьким элегантным языком.

Язык C++ продолжит развиваться в будущем, поэтому я надеюсь, что в процессе развития «C++ как разговорный» станет еще более красивым.

*Герб Стампер
Июнь 2021 года*

Дж. Гай Дэвидсон, Кейт Грегори

**Красивый C++: 30 главных правил чистого,
безопасного и быстрого кода**

Перевела с английского Л. Киселева

Руководитель дивизиона

Ю. Сергиенко

Руководитель проекта

А. Питиримов

Ведущий редактор

Н. Гринчик

Научный редактор

А. Котов

Литературный редактор

Н. Куликова

Художественный редактор

В. Мостипан

Корректоры

Е. Павлович, Н. Терех

Верстка

Г. Блинов

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,

Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 26.01.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 700. Заказ 0000.