
HIGHEST
АНДРЕЙ
АЛЕКСАНДРЕСКУ
Язык
программирования
D
Символ

16 лет вместе
с профессионалами

The D Programming Language

Andrei Alexandrescu

Addison-Wesley

HIGH TECH

Язык программирования D

Андрей Александреску

Санкт-Петербург — Москва
2012

Серия «High tech»

Андрей Александреску

Язык программирования D

Перевод Н. Данилиной

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научные редакторы	*И. Степанов*
	В. Пантелеев
Редактор	*Т. Темкина*
Корректор	*О. Макарова*
Верстка	*Д. Орлова*

Александреску А.

Язык программирования D. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 536 с., ил.

ISBN 978-5-93286-205-6

D – это язык программирования, цель которого – помочь программистам справиться с непростыми современными проблемами разработки программного обеспечения. Он создает все условия для организации взаимодействия модулей через точные интерфейсы, поддерживает целую федерацию тесно взаимосвязанных парадигм программирования (императивное, объектно-ориентированное, функциональное и метапрограммирование), обеспечивает изоляцию потоков, модульную безопасность типов, предоставляет рациональную модель памяти и многое другое.

«Язык программирования D» – это введение в D, автору которого можно доверять. Это книга в фирменном стиле Александреску – она написана неформальным языком, но без лишних слов и не в ущерб точности. Андрей рассказывает о выражениях и инструкциях, о функциях, контрактах, модулях и о многом другом, что есть в языке D. В книге вы найдете полный перечень средств языка с объяснениями и наглядными примерами; описание поддержки разных парадигм программирования конкретными средствами языка D; информацию о том, почему в язык включено то или иное средство, и советы по их использованию; обсуждение злободневных вопросов, таких как обработка ошибок, контрактное программирование и параллельные вычисления.

Книга написана для практикующего программиста, причем она не просто знакомит с языком – это настоящий справочник полезных методик и идиом, которые облегчат жизнь не только программиста на D, но и программиста вообще.

ISBN 978-5-93286-205-6
ISBN 978-0-321-63536-5 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 16.04.2012. Формат 70×100 1/16.
Печать офсетная. Объем 33,5 печ. л.

Отпечатано в цифровой типографии «Буки Веди» на оборудовании Konica Minolta
ООО «Ваш полиграфический партнер», ул. Ильменский пр-д, д. 1, корп. 6
Тел.: (495) 926-63-96, www.bukivedi.com, info@bukivedi.com. Заказ № 1298.

Оглавление

Об авторе

Андрей Александреску – автор неофициального термина «современный C++». Сегодня под этим понимают множество полезных стилей и идей программирования на C++. Книга Александреску «Современное проектирование на C++: обобщенное программирование и прикладные шаблоны проектирования» (Вильямс, 2004) полностью изменила методику программирования на C++ и оказала огромное влияние не только на непосредственную работу на C++, но и на другие языки и системы. В соавторстве с Гербом Саттером Андрей написал «Стандарты программирования на C++: 101 правило и рекомендация» (Вильямс, 2005). Благодаря разработанным им многочисленным библиотекам и приложениям, а также исследовательской работе в области машинного обучения и обработки естественных языков, Андрей снискал уважение как практиков, так и теоретиков.

С 2006 года он стал правой рукой Уолтера Брайта – автора языка программирования D и первого, кто взялся за его реализацию. Именно Андрей Александреску предложил многие важные средства D и создал большую часть стандартной библиотеки D. Все это позволило ему написать эту авторитетную книгу о новом языке. Вашингтонский университет присудил Андрею степень доктора философии компьютерных наук, а университет «Политехника» в Бухаресте – степень бакалавра электротехники. Он также является научным сотрудником Facebook.

Предисловие Уолтера Брайта

Когда-то в научно-фантастическом романе мне встретилась строка, где было сказано: ученый бесстрашно заглянет во врата ада, если думает, что это позволит расширить познания в интересующей его области. В одной-единственной фразе заключена сущность того, что значит быть ученым. Радость открытия, жажда познания хорошо видны на видеозаписях и в работах физика Ричарда Фейнмана, чей энтузиазм заражает и завораживает.

Не будучи сам ученым, я понимаю, что движет такими людьми. Мне, инженеру, тоже знакома радость творения, создания чего-то из ничего. Одна из моих любимых книг – «Братья Райт как инженеры» Уольда[1]. Это хроника пути, который шаг за шагом прошли братья Райт, одну за другой разрешая проблемы полета, чтобы затем отдать все эти знания созданию летательной машины.

Мои ранние увлечения, суть которых отражают слова с первых страниц «Руководства для любителей фейерверков» Бринли[2] – «восхищение и зачарованность всем, что горит и взрывается», – позже выросли в желание создавать то, что работает быстрее и лучше.

Но производство мощных машин – дорогое удовольствие. И тогда я открыл для себя компьютеры. Чудесное и притягательное свойство компьютеров – это легкость, с которой можно творить. Не нужен ни суперзавод за миллиард долларов, ни мастерская, ни даже отвертка. Имея всего лишь недорогой компьютер, можно создавать целые миры.

Вот я и начал создавать воображаемые миры на компьютере. Первым стала игра Empire: Wargame of the Century[3]. Компьютеры тогда были недостаточно мощны, чтобы можно было нормально играть в нее, и я заинтересовался оптимизацией программ. Это привело к изучению компиляторов, которые генерируют код, и, естественно, к высокомерному

---

[1] Quentin R. Wald «The Wright Brothers as Engineers: an Appraisal and Flying with the Wright Brothers, one Man's Experience», 1999.

[2] Bertrand R. Brinley «Rocket Manual for Amateurs», Ballantine, 1968.

[3] Одна из первых графических компьютерных стратегических игр, оказавшая большое влияние на дальнейшее развитие игр этого жанра, в частности Civilisation. См. *http://www.classicempire.com/*. – *Прим. пер.*

«я могу написать компилятор получше этого». Влюбившись в язык C, я захотел создать компилятор и для него. И за пару лет, работая неполный день, без труда справился с этим. Затем мое внимание привлек язык Бьёрна Страуструпа C++, и я решил, что смогу дополнить компилятор C соответствующими возможностями за пару месяцев (!).

Спустя десять лет я все еще работал над этим. В процессе реализации я изучил язык во всех тонкостях. Для поддержки обширной пользовательской базы необходимо было хорошо понимать, как воспринимают язык другие люди, и знать, что работает, а что нет. Я не могу пользоваться чем-то и не думать, как можно это улучшить. В 1999 году я решил воплотить свои идеи. Началось с языка программирования Mars. Однако мои коллеги стали называть его D – сначала в шутку, но потом имя прижилось. Так и появился на свет язык программирования D.

К моменту написания этого текста языку D исполнилось уже десять лет и у него появилось новое, более значительное воплощение, которое иногда называют D2. Вместо единственного человека, не отрывающегося от клавиатуры, над языком D теперь трудится целое всемирное сообщество разработчиков, которые занимаются всеми аспектами языка и поддерживают экосистему библиотек и инструментов.

Сам язык (которому посвящена эта книга) эволюционировал от скромных основ до очень мощного языка, виртуозно решающего задачи программирования разными способами. Насколько мне известно, в D остроумно и оригинально сочетаются несколько парадигм программирования: императивное, объектно-ориентированное, функциональное и метапрограммирование.

Первое, что приходит в голову после подобного заявления: такой язык не может быть простым. И в самом деле, D – непростой язык. Но, думаю, не стоит судить о языке по его сложности. Гораздо полезнее задаться вопросом о том, как выглядят программные решения на этом языке. Просты ли, изящны ли программы на D – или сложны и бестолковы?

Один мой коллега с богатым производственным опытом заметил, что IDE[1] – необходимый для программирования инструмент, потому что позволяет одним щелчком мыши сгенерировать сотни строк стандартного кода. При использовании языка D нет острой потребности в IDE, поскольку, вместо того чтобы полагаться на фокусы генерации «заготовок» разного рода «помощниками», D исключает саму идею стандартных заготовок, применяя интроспекцию и собственные возможности генерации кода. Программист уже не увидит стандартный код. О присущей программам сложности заботится язык, а не IDE.

Предположим, кто-то хочет написать программу в стиле объектно-ориентированного программирования (ООП) на простом языке без встроенной поддержки этой парадигмы. Он может сделать это ценой ужас-

[1] IDE (Integrated Development Environment) – интегрированная среда разработки.– *Прим. пер.*

ных и почти всегда напрасных усилий. Но если более сложный язык уже поддерживает ООП, писать ООП-программы легко и изящно. Язык сложнее, а код пользователя – проще. Вот куда стоит двигаться.

Чтобы легко и изящно писать код, реализующий широкий спектр задач, необходим язык с поддержкой нескольких разных парадигм программирования. Грамотно написанный код должен красиво смотреться, и, как ни странно, красивый код – это зачастую правильный код. Не знаю, чем обусловлена эта взаимосвязь, но обычно так и есть. Это как с самолетами: тот, что хорошо выглядит, как правило, и летает хорошо. То есть средства языка, позволяющие выражать алгоритмы красиво, – скорее всего, хорошие средства.

Однако только простоты и изящества написания кода мало для того, чтобы назвать язык программирования хорошим. Сегодня программы быстро растут в объеме, и конца этому не видно. С такими объемами для обеспечения корректности программ все менее целесообразно полагаться на знания и опыт программиста и традиционные способы проверки работоспособности кода. Все более стоящим кажется подход, когда выявление ошибок гарантирует машина. Здесь D может похвастаться множеством стратегий, применяя которые, программист получит такие гарантии. Эти средства включают контракты, безопасность памяти, различные атрибуты функций, свойство неизменности, защиту от «угона имен» (hijack)[1], ограничители области видимости, чистые функции, юнит-тесты и изоляцию данных при многопоточном программировании.

Нет, мы не забыли о производительности! Несмотря на многочисленные высказывания о том, что вопрос быстродействия больше не актуален, и на то, что компьютеры работают в тысячу раз быстрее, чем когда я писал свой первый компилятор, потребность в более быстрых программах, очевидно, не снизится никогда. D – это язык для системного программирования. Что это значит? В двух словах, это значит, что на D можно писать операционную систему, равно как и код приложений и драйверов устройств. С более технической точки зрения это значит, что программы на D имеют доступ ко всем возможностям машины. То есть можно использовать указатели, совмещать указатели и выполнять над ними арифметические операции, обходить систему типизации и даже писать код прямо на ассемблере. Нет ничего, что программисту на D было бы полностью недоступно. Например, реализация сборщика мусора для самого языка D написана полностью на D.

---

[1] Суть проблемы следующая: программист обращается к какому-то символу (функции, классу и т. п.) по имени, но вследствие перегрузки функций по типам аргументов (из-за переопределения методов в дочерних классах) возникает спорная ситуация, и компилятор неявно обращается к некоторому символу – возможно, не тому, который нужен программисту. Компилятор D в неоднозначных ситуациях генерирует ошибку. См. *http:// dlang.org/hijack.html*. – *Прим. науч. ред.*

Минуточку. Разве такое возможно? Каким образом язык может одновременно предоставлять и немыслимые гарантии безопасности, и неподвластные никакому контролю операции с указателями? Ответ в том, что гарантии этого типа основаны на используемых конструкциях языка. Например, с помощью атрибутов функций и конструкторов типов можно предотвратить ошибки в режиме компиляции. Контракты и инварианты предоставляют гарантии корректности работы программы во время исполнения.

Большинство качеств D в той или иной форме когда-то уже появлялись в других языках. Взятые по отдельности, они не оправдывают появление нового языка. Но их комбинация – это больше, чем просто сумма частей. И комбинация D позволяет ему претендовать на звание привлекательного языка с изящными и эффективными средствами для решения необычайно широкого круга задач программирования.

Андрей Александреску известен оригинальными идеями, сформировавшими основное направление программистской мысли (см. его новаторскую книгу «Современное проектирование на C++»). Андрей примкнул к команде разработчиков языка D в 2006 году. Его вклад – серьезная теоретическая база по программированию, а также неиссякаемый поток инновационных решений проблем программного проектирования. D2 сформировался в основном благодаря ему, а эта книга во многом развивалась совместно с D. Одно из замечательных свойств написанного им о D в том, что это сочинение – не простое перечисление фактов. Это ответы на вопросы, *почему* были выбраны те или иные проектные решения. Зная, по каким причинам язык стал именно таким, гораздо легче и быстрее понять его и начать программировать на нем.

В книге Андрей иллюстрирует эти причины, решая с помощью D множество фундаментальных задач программирования. Этим он показывает не только как работает D, но и почему он работает, и как его использовать.

Надеюсь, вы получите столько же удовольствия от программирования на D, сколько я получил от работы над ним. Страницы книги Андрея просто излучают восхищение языком. Думаю, вам понравится!

Уолтер Брайт

Январь 2010 г.

Предисловие Скотта Мейерса

Как ни крути, C++ невероятно успешен. Но даже самые страстные поклонники языка не будут отрицать, что управлять этим зверем непросто. Сложность C++ повлияла на структуру самых популярных его преемников – Java и C#. Оба стремились избежать сложности своего предшественника, предоставив его основные возможности в более удобной для использования форме.

Снижение сложности велось по двум главным направлениям. Одно из них – отказ от «сложных» средств языка. Например, управление памятью вручную (единственный способ, доступный пользователям C++) было заменено «сбором мусора». Считалось, что выгоды от применения шаблонов никогда не оправдают соответствующих затрат. Поэтому ранние версии Java и C# не включали ничего похожего на поддержку обобщенного программирования в C++.

Второе направление снижения сложности подразумевало замену «сложных» средств C++ сходными, но более простыми для понимания конструкциями. Множественное наследование C++ превратилось в простое наследование плюс интерфейсы. Современные версии Java и C# поддерживают шаблоноподобные обобщенные классы, которые проще шаблонов C++.

Эти языки-потомки претендовали на нечто большее, чем просто менее сложно делать то же, что и C++. Оба определяли виртуальные машины, добавляли поддержку рефлексии и предоставляли обширные библиотеки, позволяющие многим программистам сосредоточиться не на написании нового кода, а на «склеивании» уже имеющихся компонентов. Результат – C-подобные языки для продуктивного программирования. Если требуется быстро создать программный продукт, более или менее соответствующий комбинации готовых элементов – а большинство программного обеспечения попадает в эту категорию, – Java и C# будут лучшим выбором по сравнению с C++.

Но C++ и не предназначен для скоростной разработки – это язык для *системного* программирования. Он создавался как альтернатива C по возможностям «общения» с аппаратным обеспечением (таким как драйверы и встроенные системы), напрямую использующая библиотеки и структуры данных C (например в унаследованных системах) и работающая на пределе производительности аппаратного обеспечения. На самом де-

ле, нет ничего парадоксального в том, что узкие места виртуальных машин Java и C# написаны на C++. Реализация высокопроизводительных виртуальных машин – задача языка для *системного* программирования, а не прикладного.

Цель D – стать наследником C++ в области системного программирования. Как и Java с C#, D стремится избежать сложности C++, поэтому он отчасти задействует те же техники. Сборке мусора – «добро пожаловать», ручному управлению памятью – «до свиданья»[1]. Простому наследованию и интерфейсам – да, множественному наследованию – нет. Вот и все сходство, дальше D идет уже собственной дорогой.

Она начинается с выявления функциональных изъянов C++ и их восполнения. Текущая версия C++ не поддерживает Юникод, его новая версия (C++11), находящаяся в стадии разработки, также предоставляет очень ограниченную поддержку этой кодировки. D поддерживает Юникод с момента своего появления. Как современный C++, так и C++0x не предоставляют ни средства для работы с модулями (в том числе для их тестирования), ни инструментарий для реализации парадигмы контрактного программирования, ни «безопасные» подмножества (где невозможны ошибки при работе с памятью). D предлагает все вышеперечисленное, не жертвуя при этом способностью генерировать высококачественный машинный код.

Там, где C++ одновременно и мощный, и сложный, D пытается быть не менее мощным, но более простым. Любители шаблонного метапрограммирования на C++ продемонстрировали, насколько важна технология вычислений на этапе компиляции, но, для того чтобы использовать их, им пришлось прыгать через горящие обручи синтаксиса. D предлагает те же возможности, избавляя от лингвистических мучений. В C++ вы знаете, как написать функцию, но при этом не имеете ни малейшего понятия о том, как написать соответствующую функцию, вычисляемую на этапе компиляции. А в языке D, зная, как написать функцию, вы уже точно знаете, как написать ее вариант времени компиляции, поскольку код тот же самый.

Один из самых интересных моментов, где D расходится со своими собратьями-наследниками C++, – подход к параллельным вычислениям при многопоточном программировании. Ввиду того что неверно синхронизированный доступ к разделяемым данным («гонки за данными») – это западня, угодить в которую легко, а выбраться сложно, D переворачивает традиционные представления с ног на голову: по умолчанию данные не разделяются между потоками. По мнению разработчиков D, благодаря глубоким иерархиям кэшей в современном аппаратном обеспечении память все равно зачастую реально не разделяется между ядра-

[1] На самом деле, тут все зависит от желания. Как и подобает языку для системного программирования, D позволяет вручную управлять памятью, если вы действительно этого хотите.

ми и процессорами, так зачем по умолчанию предлагать разработчикам абстракцию, которая не просто фиктивна, но еще и чревата ошибками, с трудом поддающимися отладке?

Все это (и не только) превращает D в достойный внимания экземпляр среди наследников C и является веским доводом к прочтению этой книги. Тот факт, что ее автор – Андрей Александреску, только усиливает доводы «за». Как один из проектировщиков D, реализовавший значительную часть его библиотеки, Андрей знает D лучше кого бы то ни было. И, конечно, он может описать этот язык программирования, а кроме того, еще и объяснить, *почему* D стал именно таким. Актуальные средства языка были включены в него намеренно, а те, которых пока нет, отсутствуют тоже не без причины. Андрей – один из немногих, кто способен осветить все эти вопросы.

И освещает он их на редкость увлекательно. Посреди ненужного, казалось бы, «лирического отступления» (которое на самом деле является станцией на пути к тому пункту назначения, куда вас хотят доставить) Андрей успокаивает: «Догадываюсь, что сейчас вы задаетесь вопросом, имеет ли все это отношение к вычислениям во время компиляции. Ответ: имеет. Прошу немного терпения». Понимая, что диагностические сообщения сборщика далеко не интуитивно понятны, Андрей замечает: «Если вы забыли написать `--main`, не волнуйтесь: компоновщик тут же витиевато напомнит вам об этом на своем родном языке – зашифрованном клингонском[1]». Даже ссылки на другие источники у Александреску выглядят по-особому. Вам не просто дают номер, под которым работа Уодлера «Доказательства – это программы» числится в списке литературы, а предлагают «прочесть увлекательную монографию „Доказательства – это программы“ Уодлера». Классический труд Фридла «Регулярные выражения»[2] не просто рекомендуется к прочтению, а «горячо рекомендуется».

И разумеется, в этой книге о языке программирования много примеров исходного кода, судя по которым Андрей – отнюдь не заурядный автор. Вот определенный им прототип для функции поиска[3]:

```
bool find(int[] haystack, int needle);
```

Это книга знающего автора об интересном языке программирования. Уверен, вы не пожалеете, что прочли ее.

Скотт Мейерс
Январь 2010 г.

---

[1] Клингонский язык – искусственный язык, специально придуманный для клингонов (расы воинов) – персонажей кинофильмов, сериалов, книг и компьютерных игр о вымышленной вселенной «Звездный путь». Существующие естественные языки он напоминает весьма слабо. – *Прим. пер.*

[2] Фридл «Регулярные выражения», 3-е издание, Символ-Плюс, 2008.

[3] Если перевести идентификаторы, код примет вид: `bool найти(int[] стог сена, int иголка);`. – *Прим. пер.*

Предисловие научных редакторов перевода

Хотя язык D существует уже более десяти лет, русскоязычных ресурсов по нему очень мало. По сути, это несколько статей в Интернете. Поэтому данная книга, пожалуй, – первый источник достоверной и полной информации об этом языке.

Автор книги хотел отразить язык таким, каким он «должен быть». Некоторые аспекты языка на момент написания книги еще не были реализованы в компиляторах, другие возможности, предоставляемые компиляторами, напротив, не попали в книгу. Автор справедливо полагал, что читатель, желающий ознакомиться с возможностями конкретной реализации компилятора, может обратиться к документации (например, документация для эталонного компилятора `dmd` размещена на сайте *d-programming-language.org*). К сожалению, на момент выхода книги эта документация еще не была переведена на русский язык.

С другой стороны, нам бы хотелось, чтобы читатель смог найти в этой книге не только общее описание языка, но и большую часть информации, необходимой для его практического применения. Поэтому редакция взяла на себя смелость, с согласия автора, дополнить книгу описанием некоторых значительных аспектов языка, не освещенных автором, объяснив причину их отсутствия в оригинале.

Кроме того, за те два года, что прошли с момента выхода оригинальной версии этой книги, язык претерпел некоторые изменения. Например, убраны восьмеричные числовые литералы, запрещен неявный переход к следующей метке `case` конструкции `switch`. Эти изменения мы также постарались отразить в переводе.

Следует пояснить перевод некоторых терминов.

Первый неоднозначный момент – перевод слова «character». В подавляющем большинстве издаваемых сейчас книг это слово переводится как «символ». Между тем данный перевод не вполне корректен. Со времен книги Гриса[1] о конструировании компиляторов в русском языке символом считается синтаксическая единица (symbol). Например, `double`, `main`

---

[1] Грис Д. «Конструирование компиляторов для цифровых вычислительных машин». – М.: Мир, 1975.

и оператор ++ – это символы. В учебнике информатики Бауэра и Гооза[1] символ определен как знак (литера, буква алфавита) со смыслом. Называть символом литеру – некорректно. Мы использовали слово «символ» в значении «symbol», или «знак со смыслом» (например, символ перевода строки). Термин же «character» мы перевели как «знак». Наблюдается также некоторый конфликт термина «знак» в смысле «буква», или «литера», и термина «знак» (sign) в математическом смысле (+ или -). Но, как правило, из контекста понятно, в каком смысле употребляется слово «знак». Например, тип `char` мы называем знаковым, или литерным типом, имея в виду, что в переменной этого типа может храниться знак алфавита, а не число со знаком. К слову сказать, в D тип `char` предназначен только для хранения знака (или части знака) в кодировке UTF-8 (и, являясь интегральным типом, в математическом смысле он всегда беззнаковый, как `unsigned char` в C). Для представления же однобайтного целого числа D вводит два дополнительных типа – `byte` (со знаком) и `ubyte` (без знака). То есть под «знаковым типом» мы понимаем `char`, `wchar` или `dchar`, а тип `int` называем целым типом, или целым типом со знаком.

Второй аспект – перевод слова «statement». Во многих книгах такие вещи, как `if`, `switch`, `while`, называют операторами. В книге про D такой перевод неуместен, так как язык предоставляет возможность перегрузки операторов (operator overloading) +, *, % и так далее для пользовательских типов, и неизбежна путаница с этими операторами. Поэтому мы переводим «statement» как «инструкция».

Напоследок несколько слов о самой книге. Эту книгу стоит прочитать всем. Если вы уже знакомы с D, эта книга поможет лучше понять, почему этот язык устроен именно так, и научиться использовать его с наибольшей отдачей. Если вы хотите узнать новый для себя язык, эта книга – то, что вам нужно. Поверьте, изучение D стоит потраченного времени, и хотя этот язык еще молод и находится в процессе развития, уже сейчас его можно использовать для решения серьезных задач. Причем это решение будет одновременно элегантным и эффективным.

И наконец, если вы просто ищете что-нибудь почитать, смело берите эту книгу. Неповторимый стиль изложения и масса интересных фактов делают чтение легким и увлекательным. Эта книга – для вас.

Игорь Степанов и Владимир Пантелеев

[1] Бауэр Ф. Л., Гооз Г. «Информатика. Вводный курс: в 2-х ч.» – Пер. с нем.– М.: Мир, 1990.

Введение

Обретая силу в простоте, язык программирования порождает красоту.

Поиск компромисса при противоречивых требованиях – сложная задача, для решения которой создателю языка требуется не только знание теоретических принципов и практической стороны дела, но и хороший вкус. Проектирование языка программирования – это последняя ступень мастерства в разработке программ.

D – это язык, который последовательно старается правильно действовать в пределах выбранных им ограничений, таких как доступ системного уровня к вычислительным ресурсам, высокая производительность и синтаксическая простота, к которой стремятся все произошедшие от C языки. Стараясь правильно действовать, D порой поступает традиционно – как другие языки, а порой ломает традиции с помощью свежего, инновационного решения. Иногда это приводило к пересмотру принципов, которым, казалось, D никогда не изменит. Например, большие фрагменты программного кода, а то и целые программы, могут быть написаны с помощью хорошо определенного, не допускающего ошибок памяти «безопасного подмножества» D. Ценой небольшого ограничения доступа на системном уровне приобретается огромное преимущество при отладке программ.

D заинтересует вас, если для вас важны следующие аспекты:

- *Производительность.* D – это язык для системного программирования. Его модель памяти, несмотря на сильную типизацию, совместима с моделью памяти C. Функции на D могут вызывать функции на C, а функции на C могут использовать функции D без каких-либо промежуточных преобразований.
- *Выразительность.* D нельзя назвать небольшим, минималистичным языком, но его удельная мощность достаточно велика. Он позволяет определять наглядные, не требующие объяснений инструкции, точно моделирующие сложные реалии.
- *«Крутящий момент».* Любой лихач-«самоделкин» скажет вам, что мощность еще не все – было бы где ее применить. На одних языках лучше всего пишутся маленькие программы. Синтаксические излишества других оправдываются только начиная с определенного объема программ. D одинаково эффективно помогает справляться

и с короткими сценариями, и с большими программами, и для него отнюдь не редкость целый проект, органично вырастающий из простенького скрипта в единственном файле.

- *Параллельные вычисления.* Подход к параллельным вычислениям – несомненное отличие D от похожих языков, отражающее разрыв между современными аппаратными решениями и архитектурой компьютеров прошлого. D покончил с проклятьем неявного разделения памяти (хотя и допускает статически проверенное, явно заданное разделение) и поощряет независимые потоки, которые «общаются» друг с другом посредством сообщений.
- *Обобщенное программирование.* Идея обобщенного кода, манипулирующего другим кодом, была впервые реализована в мощных макросах Лиспа, затем в шаблонах C++, обобщенных классах Java и схожих конструкциях других языков. D также предлагает невероятно мощные механизмы обобщенного и порождающего программирования.
- *Эклектизм.* D подразумевает, что каждая парадигма программирования ориентирована на свою задачу разработки. Поэтому он предполагает высокоинтегрированный объединенный стиль программирования, а не Единственно Верный Подход.
- *«Это мои принципы. А если они вам не нравятся, то у меня есть и другие»*[1]. D старается всегда следовать своим принципам устройства языка. Иногда они идут вразрез с соображениями сложности реализации и трудностей использования и, главное, с человеческой природой, которая не всегда находит скрытую логику здравой и интуитивно понятной. В таких случаях все языки полагаются на собственное бесконечно субъективное понимание баланса, гибкости и – особенно – хорошего вкуса. На мой взгляд, D как минимум неплохо смотрится на фоне других языков, разработчикам которых приходилось принимать решения того же плана.

Кому адресована эта книга

Предполагается, что вы программист. То есть знаете, как решить типичную задачу программирования с помощью языка, на котором вы пишете. Неважно, какой конкретно это язык. Если вы знаете один из языков, произошедших от Алгола (C, C++, Java или C#), то будете иметь некоторое преимущество перед другими читателями – синтаксис сразу покажется знакомым, а риск встретить «мнимых друзей» (одинаковый синтаксис с разной семантикой) будет минимальным. (Особенно это касается случаев, когда вы вставляете кусок кода на C в D-файл. Он либо скомпилируется и будет делать то же самое, либо не скомпилируется вообще.)

Книга, знакомящая с языком, была бы скучной и неполной, если бы не объясняла, зачем в язык включены те или иные средства, и не показы-

[1] Афоризм американского комика Граучо Маркса. – *Прим. ред.*

вала наиболее рациональные пути использования этих средств для решения конкретных задач. Эта книга логично обосновывает добавление в язык всех неочевидных средств и старается показать, почему не были выбраны лучшие, на первый взгляд, проектные решения. Некоторые альтернативы требуют необоснованно высоких затрат на реализацию, плохо взаимодействуют с другими средствами языка, имеющими больше прав на существование, обладают скрытыми недостатками, которые не видны в коротких и простых примерах, или просто недостаточно мощны для того, чтобы что-то значить. Важнее всего то, что разработчики языка могут совершать ошибки так же, как и все остальные люди, поэтому, пожалуй, лучшие проектные решения – те, которых никто никогда не видел.

Структура книги

Глава 1 – это бодрящая прогулка с целью знакомства с основами языка. На этом этапе не все детали полностью видны, но вы сможете почувствовать язык и научиться писать на нем простейшие программы. Главы 2 и 3 – необходимое перечисление выражений и инструкций языка соответственно. Я попытался скрасить неизбежную монотонность подробного описания, подчеркнув детали, отличающие D от других традиционных языков. Надеюсь, вам будет легко читать эти главы подряд, а также возвращаться к ним за справкой. Таблицы в конце этих глав – это «шпаргалки», интуитивно понятные краткие справочники.

В главе 4 описаны встроенные типы: массивы, ассоциативные массивы и строки. Массив можно представить себе как указатель с аварийным выключателем. Массивы в D – это средство, обеспечивающее безопасность памяти и позволяющее вам наслаждаться языком. Строки – это массивы знаков Юникода в кодировке UTF. Повсеместная поддержка Юникода в языке и стандартной библиотеке позволяет корректно и эффективно обрабатывать строки.

Прочитав первые четыре главы, вы сможете на основе предоставляемых языком абстракций писать простые программы вроде сценариев. Последующие главы знакомят с абстракциями-блоками. Глава 5 объединяет описание различных видов функций: параметризированных функций режима компиляции (шаблоны функций) и функций, вычисляемых во время компиляции. Обычно такие вопросы рассматриваются в более «продвинутых» главах, но в D работать с этими средствами достаточно просто, так что раннее знакомство с ними оправданно.

В главе 6 обсуждается объектно-ориентированное программирование на основе классов. Как и раньше, здесь органично и комплексно подается информация о параметризированных классах. Глава 7 знакомит с дополнительными типами, в частности с типом `struct`, позволяющим, обычно совместно с классами, эффективно создавать абстракции.

Следующие четыре главы описывают довольно специализированные, обособленные средства. Глава 8 посвящена квалификаторам типов. Квалификаторы надежно гарантируют от ошибок, что одинаково ценно как для однопоточных, так и для многопоточных приложений. В главе 9 рассмотрены модели обработки исключительных ситуаций. В главе 10 представлен мощный инструментарий D, реализующий парадигму контрактного программирования. Этот материал намеренно вынесен в отдельную главу (а не включен в главу 9) в попытке развеять миф о том, что обработка ошибок и контрактное программирование – практически одно и то же. В главе 10 как раз и объясняется, почему это не так.

В главе 11 вы найдете информацию и рекомендации по построению больших программ из компонентов, а также небольшой обзор стандартной библиотеки D. В главе 12 рассмотрены вопросы перегрузки операторов, без которой серьезно пострадали бы многие абстракции, например комплексные числа. Наконец, в главе 13 освещен оригинальный подход D к многопоточному программированию.

Краткая история

Как бы сентиментально это ни звучало, D – дитя любви. Когда-то в 1990-х Уолтер Брайт, автор компиляторов для C и C++, решил, что больше не хочет работать над ними, и задался целью определить язык, каким, по его мнению, «он должен быть». Многие из нас в тот или иной момент начинают мечтать об определении Правильного Языка; к счастью, Уолтер уже обладал значительной частью инфраструктуры: генератором кода (backend), компоновщиком, а главное – широчайшим опытом построения языковых процессоров. Благодаря этому опыту перед Уолтером открылась интересная перспектива. По какому-то таинственному закону природы плохо спроектированная функциональность языка проявляется в логически запутанной реализации компилятора, как отвратительный характер Дориана Грея проявлялся на его портрете. Проектируя свой новый язык, Уолтер планомерно старался избежать таких патологий.

Едва зарождающийся тогда язык был схож по духу с C++, поэтому программисты называли его просто D, несмотря на первоначальную попытку Уолтера даровать ему титул «Марса». По причинам, которые вскоре станут очевидными, назовем этот язык D1. Страсть и упорство, с которыми Уолтер работал над D1 несколько лет, привлекали все больше единомышленников. К 2006 году D1 достиг уровня сильного языка, технически способного на равных соперничать с такими уже признанными языками, как Java и C++. Но к тому времени уже было ясно, что D1 никогда не станет популярным, поскольку, в отличие от других языков, он не обладал функциональной индивидуальностью, оправдывавшей его существование. И тогда Уолтер совершил дерзкий маневр: решив представить D1 в качестве этакой первой сырой версии, он перевел его в режим поддержки и приступил к разработке нового проекта – второй итерации языка, не обязанной поддерживать обратную совместимость.

Пользователи текущей версии D1 по-прежнему выигрывали от исправления ошибок, но никаких новых возможностей D1 не предоставлял. Реализовать определение наилучшего языка было суждено языку D2, который я и называю просто D.

Маневр удался. Первая итерация показала, что достойно внимания, а чего следует избегать. Кроме того, можно было не спешить с рекламой нового языка – новые члены сообщества могли спокойно работать со стабильной, активно используемой версией D1. Поскольку процесс разработки не был ограничен ни обратной совместимостью, ни сроками, можно было спокойно оценить альтернативы развития проекта и выработать правильное направление. Чтобы еще больше облегчить разработку, Уолтер призвал на помощь коллег, в том числе Бартоша Милевски и меня. Важные решения, касающиеся взглядов D на неизменяемость, обобщенное и функциональное программирование, параллельные вычисления, безопасность и многое другое, мы принимали в долгих оживленных дискуссиях на троих в одной из кофеен Киркленда (штат Вашингтон).

Тем временем D явно перерос свое прозвище «улучшенный C++», превратившись в мощный многофункциональный язык, вполне способный оставить без работы как языки для системного и прикладного (промышленного) программирования, так и языки сценариев. Оставалась одна проблема: весь этот рост и все усовершенствование прошли никем не замеченными; подходы D к программированию были документированы очень слабо.

Книга, которая сейчас перед вами, – попытка восполнить это упущение. Надеюсь, читать ее вам будет так же приятно, как мне – писать.

Благодарности

У языка D было столько разработчиков, что я и не пытаюсь перечислить их всех. Особо выделяются участники новостной группы `digitalmars.D` из сети Usenet. Эта группа была для нас одновременно и рупором, и полигоном для испытаний, куда мы выносили на суд свои проектные решения. Кроме того, ребята из `digitalmars.D` сгенерировали множество идей по улучшению языка.

В разработке эталонной реализации компилятора `dmd`[1] Уолтеру помогало сообщество, особенно Шон Келли (Sean Kelly) и Дон Клагстон (Don Clugston). Шон переписал и усовершенствовал стандартную библиотеку, подключаемую во время исполнения (включая «сборщик мусора»). Кроме того, Келли стал автором основной части реализации библиотеки, отвечающей за параллельные вычисления. Он мастер своего дела, а значит, если в ваших параллельных вычислениях появляются ошибки, то

[1] Название компилятора языка D `dmd` расшифровывается как Digital Mars D. Digital Mars – организация, которая занимается разработкой этого компилятора. – *Прим. пер.*

они, увы, скорее всего, ваши, а не его. Дон – эксперт в математике вообще и во всех аспектах дробных вычислений в частности. Его огромный труд позволил поднять численные примитивы D на небывалую высоту. Кроме того, Дон до предела использовал способности D по генерированию кода. Как только код эталонной реализации был открыт для широкого доступа, Дон не устоял перед соблазном добавить в него что-то свое. Вот так он и занял второе место среди разработчиков компилятора `dmd`. И Шон, и Дон проявляли инициативу, выдвигая предложения по усовершенствованию спецификации D на протяжении всего процесса разработки. Последнее (но не по значению) их достоинство в том, что они чумовые хакеры. С ними очень приятно общаться как в жизни, так и виртуально. Не знаю, чем стал бы язык без них.

Что касается этой книги, я бы хотел сердечно поблагодарить всех рецензентов за отзывчивость, с которой они взялись за эту сложную и неблагодарную работу. Без них эта книга не стала бы тем, что она представляет собой сейчас (так что если она вам не нравится, пусть вас утешит то, что она могла быть гораздо хуже). Поэтому позвольте мне выразить благодарность Алехандро Арагону (Alejandro Aragón), Биллу Бакстеру (Bill Baxter), Кевину Билеру (Kevin Bealer), Тревису Бочеру (Travis Boucher), Майку Касинджино (Mike Casinghino), Альваро Кастро Кастилья (Àlvaro Castro Castilla), Ричарду Чангу (Richard Chang), Дону Клагстону, Стефану Дилли (Stephan Dilly), Кариму Филали (Karim Filali), Мишелю Фортину (Michel Fortin), Дэвиду Хелду (David B. Held), Мишелю Хелвенштейну (Michiel Helvensteijn), Бернарду Хельеру (Bernard Helyer), Джейсону Хаузу (Jason House), Сэму Ху (Sam Hu), Томасу Хьюму (Thomas Hume), Грэму Джеку (Graham St. Jack), Роберту Жаку (Robert Jacques), Кристиану Кэмму (Christian Kamm), Дэниелу Кипу (Daniel Keep), Марку Кегелу (Mark Kegel), Шону Келли, Максу Хесину (Max Khesin), Симену Хьеросу (Simen Kjærås), Коди Кёнингеру (Cody Koeninger), Денису Короскину (Denis Koroskin), Ларсу Кюллингстаду (Lars Kyllingstad), Игорю Лесику (Igor Lesik), Евгению Летучему (Eugene Letuchy), Пелле Манссону (Pelle Månsson), Миуре Масахиро (Miura Masahiro), Тиму Мэтьюсу (Tim Matthews), Скотту Мейерсу, Бартошу Милевски, Фавзи Мохамеду (Fawzi Mohamed), Эллери Ньюкамеру (Ellery Newcomer), Эрику Ниблеру (Eric Niebler), Майку Паркеру (Mike Parker), Дереку Парнеллу (Derek Parnell), Джереми Пеллетье (Jeremie Pelletier), Пабло Риполлесу (Pablo Ripolles), Брэду Робертсу (Brad Roberts), Майклу Ринну (Michael Rynn), Фою Сава (Foy Savas), Кристофу Шардту (Christof Schardt), Стиву Швайхофферу (Steve Schveighoffer), Бенджамину Шропширу (Benjamin Shropshire), Дэвиду Симше (David Simcha), Томашу Стаховяку (Tomasz Stachowiak), Роберту Стюарту (Robert Stewart), Кнуту Эрику Тайгену (Knut Erik Teigen), Кристиану Влащану (Cristian Vlăsceanu) и Леору Золману (Leor Zolman).

Андрей Александреску

Воскресенье 2 мая 2010 г.

1

Знакомство с языком D

Вы ведь знаете, с чего обычно начинают, так что без лишних слов:

```
import std.stdio;
void main() {
   writeln("Hello, world!");
}
```

В зависимости от того, какие еще языки вы знаете, у вас может возникнуть ощущение дежавю, чувство легкой благодарности за простоту, а может, и легкого разочарования из-за того, что D не пошел по стопам скриптовых языков, разрешающих использовать «корневые» (top-level) инструкции. (Такие инструкции побуждают вводить глобальные переменные, которые по мере роста программы превращаются в головную боль; на самом деле, D позволяет исполнять код не только внутри, но и вне функции `main`, хотя и более организованно.) Самые въедливые будут рады узнать, что `void main` – это эквивалент функции `int main`, возвращающей операционной системе «успех» (код 0) при успешном окончании ее выполнения.

Но не будем забегать вперед. Традиционная программа типа «Hello, world!» («Здравствуй, мир!») – вовсе не повод для обсуждения возможностей языка. Она здесь для того, чтобы помочь вам начать писать и запускать программы на этом языке. Если у вас нет никакой IDE, которая выполнит за вас сборку программы, то самый простой способ – это командная строка. Напечатав приведенный код и сохранив его в файле с именем, скажем, `hello.d`, запустите консоль и введите следующие команды:

```
$ dmd hello.d
$ ./hello
Hello, world!
$ _
```

Знаком $ обозначено приглашение консоли вашей ОС (это может быть c:\Путь\К\Папке> в Windows или /путь/к/каталогу% в системах семейства UNIX, таких как OSX, Linux, Cygwin). Применив пару известных вам приемов систем-фу, вы сможете добиться автоматической компиляции программы при ее запуске. Пользователи Windows, вероятно, захотят привязать программу rdmd.exe (которая устанавливается вместе с компилятором D) к команде Выполнить. UNIX-подобные системы поддерживают запуск скриптов в нотации «shebang»[1]. D понимает такой синтаксис: добавление строки

```
#!/usr/bin/rdmd
```

в самое начало программы в файле hello.d позволяет компилировать ее автоматически перед исполнением. Внеся это изменение, просто введите в командной строке:

```
$ chmod u+x hello.d
$ ./hello.d
Hello, world!
$ _
```

(chmod нужно ввести только один раз).

Для всех операционных систем справедливо следующее: программа rdmd достаточно «умна», для того чтобы кэшировать сгенерированное приложение. Так что фактически компиляция выполняется только после изменения исходного кода программы, а не при каждом запуске. Эта особенность в сочетании с высокой скоростью самого компилятора позволяет экономить время на запусках программы между внесением в нее изменений, что одинаково полезно как при разработке больших систем, так и при написании маленьких скриптов.

Программа hello.d начинается с инструкции

```
import std.stdio;
```

которая предписывает компилятору найти модуль с именем std.stdio и сделать его символы доступными для использования. Инструкция import напоминает препроцессорную директиву #include, которую можно встретить в синтаксисе C и C++, но семантически она ближе команде import языка Python: никакой вставки текста подключаемого модуля в текст основной программы не происходит – выполняется только простое расширение таблицы символов. Если повторно применить инструкцию import к тому же файлу, ничего не произойдет.

По давней традиции C программа на D представляет собой набор определений, рассредоточенный по множеству файлов. В числе прочего эти определения могут обозначать типы, функции, данные. В нашей первой

[1] «Shebang» (от shell bang: shell – консоль, bang – восклицательный знак), или «shabang» (# – sharp) – обозначение пути к компилятору или интерпретатору в виде #!/путь/к/программе. – *Прим. пер.*

программе определена функция main. Она не принимает никаких аргументов и ничего не возвращает, что, по сути, и означает слово void. При выполнении main программа вызывает функцию writeln (разумеется, предусмотрительно определенную в модуле std.stdio), передавая ей строковую константу в качестве аргумента. Суффикс ln указывает на то, что writeln добавляет к выводимому тексту знак перевода строки.

Следующие разделы – это стремительная поездка по Дибургу. Небольшие показательные программы дают общее представление о языке. Основная цель повествования на данном этапе – обрисовать общую картину, а не дать ряд педантичных определений. Позже все аспекты языка будут рассмотрены с должным вниманием – в деталях.

1.1. Числа и выражения

Интересовались ли вы когда-нибудь ростом иностранцев? Давайте напишем простую программу, которая переводит наиболее распространенные значения роста в футах и дюймах в сантиметры.

```
/*
  Рассчитать значения роста в сантиметрах
  для заданного диапазона значений в футах и дюймах
*/
import std.stdio;

void main() {
   // Значения, которые никогда не изменятся
   immutable inchesPerFoot = 12;
   immutable cmPerInch = 2.54;
   // Перебираем и пишем
   foreach (feet; 5    7) {
      foreach (inches; 0    inchesPerFoot) {
         writeln(feet, "'"  inches, "''\t"
            (feet * inchesPerFoot + inches) * cmPerInch;
      }
   }
}
```

В результате выполнения программы будет напечатан аккуратный список в две колонки:

```
5'0''       152.4
5'1''       154.94
5'2''       157.48

6'10''      208.28
6'11''      210.82
```

Инструкция foreach (feet; 5..7) {...} – это цикл, где определена целочисленная переменная feet, с которой последовательно связываются значения 5 и 6 (значение 7 она не принимает, так как интервал открыт справа).

Как и Java, C++ и C#, D поддерживает `/* многострочные комментарии */` и `// однострочные комментарии` (и, кроме того, документирующие комментарии, о которых позже). Еще одна интересная деталь нашей маленькой программы – способ объявления данных. Во-первых, введены две константы:

```
immutable inchesPerFoot = 12;
immutable cmPerInch = 2.54;
```

Константы, значения которых никогда не изменятся, определяются с помощью ключевого слова `immutable`. Как и переменные, константы не требуют явного задания типа: тип задается значением, которым инициализируется константа или переменная. В данном случае литерал `12` говорит компилятору о том, что `inchesPerFoot` – это целочисленная константа (обозначается в D с помощью знакомого `int`); точно так же литерал `2.54` заставляет `cmPerInch` стать константой с плавающей запятой (типа `double`). Далее мы обнаруживаем те же магические способности у определений `feet` и `inches`: они выглядят как «обычные» переменные, но безо всяких «украшений», свидетельствующих о каком-либо типе. Это не делает программу менее безопасной по сравнению с той, где типы переменных и констант заданы явно:

```
immutable int inchesPerFoot = 12;
immutable double cmPerInch = 2.54;
 ..
foreach (int feet; 5    7) {

}
```

и так далее – только меньше лишнего. Компилятор разрешает не указывать тип явно только в случае, когда можно недвусмысленно определить его по контексту. Раз уж зашла речь о типах, давайте остановимся и посмотрим, какие числовые типы нам доступны.

Целые типы со знаком в порядке возрастания размера: `byte`, `short`, `int` и `long`, занимающие 8, 16, 32 и 64 бита соответственно. У каждого из этих типов есть «двойник» без знака того же размера, названный в соответствии с простым правилом: `ubyte`, `ushort`, `uint` и `ulong`. (Здесь нет модификатора `unsigned`, как в C). Типы с плавающей запятой: `float` (32-битное число одинарной точности в формате IEEE 754), `double` (64-битное в формате IEEE 754) и `real` (занимает столько, сколько позволяют регистры, предназначенные для хранения чисел с плавающей запятой, но не меньше 64 бит; например, на компьютерах фирмы Intel `real` – это так называемое расширенное 79-битное число двойной точности в формате IEEE 754).

Вернемся к нашим целым числам. Литералы, такие как `42`, подходят под определение любого числового типа, но заметим, что компилятор проверяет, достаточно ли вместителен «целевой» тип для этого значения. Поэтому определение

```
immutable byte inchesPerFoot = 12;
```

ничем не хуже аналогичного без byte, поскольку 12 можно с таким же успехом представить 8 битами, а не 32. По умолчанию, если вывод о «целевом» типе делается по числу (как в программе-примере), целочисленные константы «воспринимаются» как int, а дробные – как double.

Вы можете построить множество выражений на D, используя эти типы, арифметические операторы и функции. Операторы и их приоритеты сходны с теми, что можно найти в языках-собратьях D: +, -, *, / и % для базовых арифметических операций, ==, !=, <, >, <=, >= для сравнений, fun(argument1, argument2) для вызовов функций и т. д.

Вернемся к нашей программе перевода дюймов в сантиметры и отметим две достойные внимания детали вызова функции writeln. Первая: во writeln передаются 5 аргументов (а не один, как в той программе, что установила контакт между вами и миром D). Функция writeln очень похожа на средства ввода-вывода, встречающиеся в языках Паскаль (writeln), C (printf) и C++ (cout). Все они (включая writeln из D) принимают переменное число аргументов (так называемые функции с переменным числом аргументов). Однако в D пользователи могут определять собственные функции с переменным числом аргументов (чего нет в Паскале), которые всегда типизированы (в отличие от C), без излишнего переопределения операторов (как это сделано в C++). Вторая деталь: наш вызов writeln неуклюже сваливает в кучу информацию о форматировании и форматируемые данные. Обычно желательно отделять данные от представления. Поэтому давайте используем специальную функцию writefln, осуществляющую форматированный вывод:

```
writefln("%s'%s''\t%s", feet, inches,
   (feet * inchesPerFoot + inches) * cmPerInch) ;
```

По-новому организованный вызов дает тот же вывод, но первый аргумент функции writefln полностью описывает формат представления. Со знака % начинаются спецификаторы формата (по аналогии с функцией printf из C): например %d – для целых чисел, %f – для чисел с плавающей запятой и %s – для строк.

Если вы использовали printf прежде, то могли бы почувствовать себя как дома, когда б не маленькая особенность: мы ведь выводим значения переменных типа int и double – как же получилось, что и те и другие описаны с помощью спецификатора %s, обычно применяемого для вывода строк? Ответ прост. Средства D для работы с переменным количеством аргументов дают writefln доступ к информации об исходных типах переданных аргументов. Благодаря такому подходу программа получает ряд преимуществ: 1) значение %s может быть расширено до «строкового представления по умолчанию для типа переданного аргумента» и 2) если не удалось сопоставить спецификатор формата с типами переданных аргументов, вы получите ошибку в чистом виде, а не загадочное поведение, присущее вызовам printf с неверно заданным форматом (не говоря уже о подрыве безопасности, возможном при вызове printf с непроверяемыми заранее форматирующими строками).

1.2. Инструкции

В языке D, как и в других родственных ему языках, любое выражение, после которого стоит точка с запятой, – это инструкция (например в программе «Hello, world!» сразу после вызова `writeln` есть ;). Действие инструкции сводится к вычислению выражения.

D – член семейства с фигурными скобками и с блочной областью видимости». Это означает, что вы можете объединять несколько команд в одну, помещая их в { и }, что порой обязательно, например при желании сделать сразу несколько вещей в цикле `foreach`. В случае единственной команды вы вправе смело опустить фигурные скобки. На самом деле, весь наш двойной цикл, вычисляющий значения роста, можно переписать так:

```
foreach (feet; 5    7)
   foreach (inches; 0    inchesPerFoot)
      writefln("%s'%s''\t%s"  feet, inches,
         (feet * inchesPerFoot + inches) * cmPerInch);
```

У пропуска фигурных скобок для одиночных инструкций есть как преимущество (более короткий код), так и недостаток – редактирование кода становится более утомительным (в процессе отладки придется повозиться с инструкциями, то добавляя, то удаляя скобки). Когда речь заходит о правилах расстановки отступов и фигурных скобок, мнения сильно расходятся. На самом деле, пока вы последовательны в своем выборе, все это не так важно, как может показаться. В качестве доказательства: стиль, предлагаемый в этой книге (обязательное заключение в операторные скобки даже одиночных инструкций, открывающая скобка на одной строке с соответствующим оператором, закрывающие скобки на отдельных строках), по типографским причинам отличается от реально применяемого автором. А раз он мог спокойно это пережить, не превратившись в оборотня, то и любой сможет.

Благодаря языку Python стал популярен иной способ отражения блочной структуры программы – с помощью отступов (чудесное воплощение принципа «форма соответствует содержанию»). Для программистов на других языках утверждение, что пробел имеет значение, – всего лишь нелепая фраза, но для тех, кто пишет на Python, это зарок. D обычно игнорирует пробелы, но он разработан с прицелом на легкость синтаксического разбора (т. е. чтобы при разборе не приходилось выяснять значения символов). А это подразумевает, что в рамках скромного «комнатного» проекта можно реализовать простой препроцессор, позволяющий использовать для выделения блоков инструкций отступы (как в Python) без каких-либо неудобств во время компиляции, исполнения и отладки программ.

Кроме того, вам должна быть хорошо знакома инструкция `if`:

```
if (‹выражение›) ‹инструкция1› else ‹инструкция2›
```

Чисто теоретический вывод, известный как принцип структурного программирования [10], гласит, что все алгоритмы можно реализовать с помощью составных инструкций, if-проверок и циклов а-ля for и foreach. Разумеется, любой адекватный язык (как и D) предлагает гораздо больше, но мы пока постановим, что с нас довольно и этих инструкций, и двинемся дальше.

1.3. Основы работы с функциями

Оставим пока в стороне обязательное определение функции main и посмотрим, как определяются другие функции на D. Определение функции соответствует модели, характерной и для других Алгол-подобных языков: сначала пишется возвращаемый тип, потом имя функции и, наконец, заключенный в круглые скобки список формальных аргументов, разделенных запятыми. Например, определение функции с именем pow, которая принимает значения типа double и int, а возвращает double, записывается так:

```
double pow(double base, int exponent) {

}
```

Каждый параметр функции (base и exponent в данном примере) кроме типа может иметь необязательный *класс памяти* (*storage class*), определяющий способ передачи аргумента в функцию при ее вызове[1].

По умолчанию аргументы передаются в pow по значению. Если перед типом параметра указан класс памяти ref, то параметр привязывается напрямую к входному аргументу, так что изменение параметра непосредственно отражается на значении, полученном извне. Например:

```
import std.stdio;

void fun(ref uint x, double y) {
   x = 42;
   y = 3.14;
}
void main() {
   uint a = 1;
   double b = 2;
   fun(a, b);
   writeln(a, " "  b);
}
```

Эта программа печатает 42 2, потому что x определен как ref uint, то есть когда значение присваивается x, на самом деле операция проводится с a. С другой стороны, присваивание значения переменной y никак

[1] В этой книге под «параметром» понимается значение, используемое внутри функции, а под «аргументом» — значение, передаваемое в функцию извне.

не скажется на b, поскольку y – это внутренняя копия в распоряжении функции fun.

Последние «украшения», которые мы обсудим в этом кратком введении, – это in и out. Попросту говоря, in – данное функцией «обещание» только смотреть на параметр, не «трогая» его. Указание out в определении параметра функции действует сходно с ref, с той поправкой, что параметр принудительно инициализируется своим значением по умолчанию при «входе» в функцию. (Для каждого типа T определено начальное значение, обозначаемое как T.init. Пользовательские типы могут определять собственное значение по умолчанию.)

О функциях можно еще долго рассказывать. Можно передавать функции другим функциям, встраивать одну в другую, разрешать функции сохранять свою локальную среду (полнофункциональная синтаксическая клауза), создавать анонимные функции (лямбда-функции), с удобством манипулировать ими и еще множество дополнительных «вкусностей». Со временем мы доберемся до каждой из них.

1.4. Массивы и ассоциативные массивы

Массивы и ассоциативные массивы (которые обычно называют хеш-таблицами, или хешами) – пожалуй, наиболее часто используемые сложные структуры данных за всю историю машинных вычислений, завистливо преследуемые списками языка Лисп. Множество полезных программ не требуют ничего, кроме массива или ассоциативного массива. Так что пришло время посмотреть, как D их реализует.

1.4.1. Работаем со словарем

Для примера напишем простенькую программку, следуя такой спецификации:

> **Читать текст, состоящий из слов, разделенных пробелами, и сопоставлять каждому не встречавшемуся до сих пор при чтении слову уникальное число. Вывод организовать в виде строк формата:**
>
> идентификатор слово

Такой маленький скрипт вполне может пригодиться, когда вы захотите обработать какой-нибудь текст. Построив словарь, вы получите возможность манипулировать только числами (что дешевле), а не полновесными словами. Один из вариантов построения такого словаря – накапливать уже прочитанные слова в ассоциативном массиве, отображающем слова на целые числа. При добавлении нового соответствия достаточно убедиться, что число, связываемое со словом, уникально («железная» гарантия – просто использовать текущую длину массива, в результате чего получится последовательность идентификаторов 0, 1, 2, ...). Посмотрим, как это можно реализовать на D.

```
import std.stdio, std.string;

void main() {
   size_t [string] dictionary;
   foreach (line; stdin.byLine()) {
      // Разбить строку на слова
      // Добавить каждое слово строки в словарь
      foreach (word; splitter(strip(line))) {
         if (word in dictionary) continue;  // Ничего не делать
         auto newID = dictionary.length;
         dictionary[word.idup] = newID;
         writeln(newID, '\t' word);
      }
   }
}
```

В языке D ассоциативный массив (хеш-таблица), который значениям типа K ставит в соответствие значения типа V, обозначается как V[K]. Итак, переменная dictionary типа size_t[string] сопоставляет строкам целые числа без знака – как раз то, что нам нужно для хранения соответствий слов идентификаторам. Выражение word in dictionary истинно, если ключевое слово word можно найти в ассоциативном массиве dictionary. Наконец, вставка в словарь выполняется так: dictionary[word.idup] = newID[1].

Хотя в рассмотренном сценарии не отражается явно тот факт, что тип string – на самом деле массив знаков, это так. В общем виде динамический массив элементов типа T обозначается как T[] и может определяться различными способами:

```
int[] a = new int[20]; // 20 целых чисел, инициализированных нулями
int[] b = [ 1, 2, 3 ]; // Массив, содержащий 1, 2, и 3
```

В отличие от массивов C, массивы D «знают» собственную длину. Для любого массива arr это значение доступно как arr.length. Присваивание значения arr.length перераспределяет память, выделенную под массив. При попытке обращения к элементам массива проверяется, не выходит ли запрашиваемый индекс за границу массива. Любители рискнуть переполнением буфера могут «выдрать» указатель из массива (используя arr.ptr) и затем выполнять непроверенные арифметические операции над ним. Кроме того, если вам действительно нужно все, что может дать кремниевая пластина, есть опция компилятора, отменяющая проверку границ. Можно сказать, что к безопасности ведет путь наименьшего сопротивления. Код безопасен по умолчанию, а если поработать, можно сделать его чуть более быстрым.

---

[1] .idup – свойство любого массива, возвращающее неизменяемую (immutable) копию массива. Про неизменяемость будет рассказано позже, пока же следует знать, что ключ ассоциативного массива должен быть неизменяемым. – *Прим. науч. ред.*

Вот как можно проходить по массиву с помощью новой формы уже знакомой инструкции foreach:

```
int[] arr = new int[20];
foreach (elem; arr) {
   /* ... использовать elem.    */
}
```

Этот цикл по очереди связывает переменную elem с каждым элементом массива arr. Присваивание elem не влияет на элементы arr. Чтобы изменить массив таким способом, просто используйте ключевое слово ref:

```
// Обнулить все элементы arr
foreach (ref elem; arr) {
   elem = 0;
}
```

Теперь, когда мы знаем, как foreach работает с массивами, рассмотрим еще один полезный прием. Если в теле цикла вам потребуется индекс элемента массива, foreach может рассчитать его для вас:

```
int[] months = new int[12];
foreach (i, ref e; months) {
   e = i + 1;
}
```

Этот код заполняет массив числами от 1 до 12. Такой цикл эквивалентен чуть более многословному определению (см. ниже), использующему foreach для просмотра диапазона чисел:

```
foreach (i; 0    months.length) {
   months[i] = i + 1;
}
```

D также предлагает массивы фиксированного размера, обозначаемые, например, как int[5]. За исключением отдельных специализированных приложений, предпочтительнее использовать динамические массивы, поскольку обычно размер массива вам заранее неизвестен.

Семантика копирования массивов неочевидна: копирование одной переменной типа массив в другую не копирует весь массив; эта операция порождает лишь новую ссылку на ту же область памяти. Если вам действительно хочется заполучить копию, просто используйте свойство массива .dup:

```
int[] a = new int[100];
int[] b = a;
// ++x увеличивает на 1 значение x
++b[10];   // В b[10] теперь 1, в a[10] то же
b = a.dup; // Полностью скопировать a в b
++b[10];   // В b[10] теперь 2, а в a[10] остается 1
```

1.4.2. Получение среза массива. Функции с обобщенными типами параметров. Тесты модулей

Получение среза массива – это мощное средство, позволяющее ссылаться на часть массива, в действительности не копируя данные массива. В качестве иллюстрации напишем функцию двоичного поиска, реализующую одноименный алгоритм: получив упорядоченный массив и значение, двоичный поиск быстро возвращает логический результат, сообщающий, есть ли заданное значение в массиве. Функция из стандартной библиотеки D возвращает более информативный ответ, чем просто булево значение, но знакомство с ней придется отложить, так как для этого необходимо более глубокое знание языка. Позволим себе, однако, приподнять планку, задавшись целью написать функцию, которая будет работать не только с массивами целых чисел, но с массивами элементов любого типа, допускающего сравнение с помощью операции <. Оказывается, реализовать эту задумку можно без особого труда. Вот как выглядит функция обобщенного двоичного поиска binarySearch:

```
import std.array;

bool binarySearch(T)(T[] input, T value) {
   while (!input.empty) {
      auto i = input.length / 2;
      auto mid = input[i];
      if (mid > value) input = input[0  . i];
      else if (mid < value) input = input[i + 1    $];
      else return true;
   }
   return false;
}

unittest {
   assert(binarySearch([ 1, 3, 6, 7, 9, 15 ], 6));
   assert(!binarySearch([ 1, 3, 6, 7, 9, 15 ], 5));
}
```

Знаки (T) в сигнатуре функции binarySearch обозначают *параметр типа* с именем T. T становится псевдонимом переданного типа в теле этой функции. Затем параметр типа можно использовать в обычном списке параметров функции. При вызове binarySearch компилятор определит значение T по фактическим аргументам. Если вы хотите указать T явно (например, для надежности), то можете написать:

```
assert(binarySearch!(int)([ 1, 3, 6, 7, 9, 15 ], 6));
```

что обнаруживает возможность вызова обобщенной функции с двумя заключенными в круглые скобки последовательностями аргументов. Сначала следуют заданные во время компиляции аргументы, заключенные в !(...), а за ними – получаемые во время исполнения программы

аргументы в (...). Объединение этих двух «владений» в одно обдумывалось, но эксперименты показали, что такая унификация создает больше проблем, чем решает.

Если вы знакомы с аналогичными средствами Java, C# и C++, то, вероятно, вам сразу бросилось в глаза то, что D сделал шаг в сторону от этих языков, отказавшись применять угловые скобки < и > для обозначения аргументов, заданных во время компиляции. Это осознанное решение. Его цель – избежать горького опыта C++ (возросшее количество трудностей при синтаксическом разборе, гекатомба в виде множества специальных правил и тай-брейков плюс ко всему неясный синтаксис, осложняющий жизнь пользователя своей двусмысленностью[1]). Проблема в том, что знаки < и > являются операторами сравнения[2]. Это делает их использование в качестве разделителей очень двусмысленным, учитывая тот факт, что *внутри* этих разделителей разрешены выражения. Таким «кандидатам в разделители» очень сложно подыскать замену. Языкам Java и C# живется легче: они запрещают писать выражения внутри < и >. Однако этим они ограничивают свою расширяемость ради сомнительного преимущества. D разрешает использовать выражения в качестве аргументов, заданных во время компиляции. Было решено упростить жизнь как человеку, так и компьютеру, наделив дополнительным смыслом традиционный унарный оператор ! (используемый в логических операциях) и задействовав классические круглые скобки (которые, уверен, вы всегда сможете верно сопоставить друг другу).

Другая любопытная деталь нашей реализации бинарного поиска – употребление `auto` для запуска алгоритма, строящего предположения о типах по контексту программы: типы переменных `i` и `mid` определены из выражений, которыми они инициализируются.

В стремлении придерживаться хорошего тона при написании программ к `binarySearch` был добавлен тест модуля. Тесты модулей вводятся в виде блоков, озаглавленных ключевым словом `unittest` (файл может содержать сколько угодно конструкций `unittest`, поскольку, как известно, проверок много не бывает). Чтобы перед входом в функцию `main` запустить тест, передайте компилятору флаг `-unittest`. Хотя `unittest` кажется незначительной деталью, такая конструкция помогает соблюдать хороший стиль программирования: с ее помощью вставлять тесты так легко, что было бы странно не делать этого. Кроме того, если вы привыкли создавать программы «сверху вниз» и предпочитаете видеть сначала тест модуля, а реализацию потом, смело вставляйте `unittest` до `binarySearch`; в D семантика символов на уровне модуля никогда не зависит от их расположения относительно других символов на том же уровне.

---

[1] Если кто-то из ваших коллег прокачал самоуверенность до уровня Супермена, спросите его, что делает код `object.template fun<arg>()`, и вы увидите криптонит в действии.

[2] Усугубляет ситуацию с угловыми скобками то, что << и >> – тоже операторы.

Операция получения среза `input[a .. b]` возвращает срез массива `input` от `a` до `b`, исключая индекс `b`. Если `a == b`, будет возвращен пустой срез, а если `a > b`, генерируется исключительная ситуация. Операция получения среза не влечет за собой динамическое выделение памяти; это всего лишь создание новой ссылки на часть массива. Символ `$` внутри выражения индексации или получения среза обозначает длину массива, к которому осуществляется доступ; например, `input[0 .. $]` – это то же самое, что и просто `input`.

Повторимся: несмотря на кажущееся обилие перемещений, производимых функцией `binarySearch`, память под новые массивы не была выделена ни разу. Предложенную реализацию алгоритма ни в коей мере нельзя назвать менее эффективной по сравнению с традиционной реализацией, которую характеризует постоянное вычисление индексов. Однако, без сомнения, новая реализация проще для понимания, поскольку она оперирует меньшим количеством состояний. В свете разговора о состояниях напишем рекурсивную реализацию `binarySearch`, которая вообще не переопределяет `input`:

```
import std.array;

bool binarySearch(T)(T[] input, T value) {
   if (input.empty) return false;
   auto i = input.length / 2;
   auto mid = input[i];
   if (mid > value) return binarySearch(input[0    i], value);
   if (mid < value) return binarySearch(input[i + 1    $]], value);
   return true;
}
```

Рекурсивная реализация явно проще и концентрированнее по сравнению со своим итеративным собратом. Кроме того, она ничуть не менее эффективна, так как рекурсивные вызовы оптимизируются благодаря популярной среди компиляторов технике, известной как *оптимизация хвостовой рекурсии*. В двух словах: если функция возвращает просто вызов самой себя (но с другими аргументами), компилятор модифицирует аргументы и инициирует переход к началу функции.

1.4.3. Подсчет частот. Лямбда-функции

Поставим себе задачу написать еще одну полезную программу, которая будет подсчитывать частоту употребления слов в заданном тексте. Хотите знать, какие слова употребляются в «Гамлете» чаще всего? Тогда вы как раз там, где надо.

Следующая программа использует ассоциативный массив, сопоставляющий строки переменным типа `uint`, и имеет структуру, напоминающую структуру программы построения словаря, рассмотренной в предыдущем примере. Чтобы сделать программу подсчета частот полностью полезной, в нее добавлен простой цикл печати:

```
import std.algorithm, std.stdio, std.string;

void main() {
   // Рассчитать таблицу частот
   uint[string] freqs;
   foreach (line; stdin.byLine()) {
      foreach (word; splitter(strip(line))) {
         ++freqs[word.idup];
      }
   }
   // Напечатать таблицу частот
   foreach (key, value; freqs) {
      writefln("%6u\t%s", value, key);
   }
}
```

А теперь, скачав из Сети файл hamlet.txt[1] (который вы найдете по прямой ссылке *http://erdani.com/tdpl/hamlet.txt*) и запустив нашу маленькую программу с шекспировским шедевром в качестве аргумента, вы получите:

```
1     outface
1     come?
1     blanket,
1     operant
1     reckon
2     liest
1     Unhand
1     dear,
1     parley.
1     share.
```

И, к сожалению, обнаружите, что вывод неупорядочен: слова, которые напечатаны в первых строках, далеко не самые часто встречающиеся. Что неудивительно – для ускорения реализации примитивов ассоциативных массивов элементы в них могут храниться в любом порядке.

Для того чтобы отсортировать вывод по убыванию частоты употребления слов, вы можете просто передать вывод программы утилите sort с флажком -nr (от numerically – отсортировать по числам и reversed – в обратном порядке), но это своего рода хитрость. Чтобы добавить сортировку непосредственно в нашу программу, заменим последний цикл следующим кодом:

```
// Напечатать таблицу частот
string[] words = freqs.keys;
sort!((a, b) { return freqs[a] > freqs[b]; })(words);
foreach (word; words) {
```

[1] Этот файл содержит текст пьесы «Гамлет». – *Прим. пер.*

```
    writefln("%6u\t%s"  freqs[word], word);
}
```

Свойство .keys позволяет получить только ключи ассоциативного массива freqs в виде массива строк, под который выделяется новая область памяти. Этого не избежать, ведь нам требуется делать перестановки. Мы получили код

```
sort!((a, b) { return freqs[a] > freqs[b]; })(words);
```

который соответствует недавно рассмотренной нотации:

```
sort!(‹аргументы времени компиляции›)(‹аргументы времени исполнения›);
```

Взяв текст, заключенный в первые круглые скобки – !(...), – мы получим форму записи, напоминающую незаконченную функцию – как будто ее автор забыл о типах параметров, возвращаемом типе и о самом имени функции:

```
(a, b) { return freqs[a] > freqs[b]; }
```

Это *лямбда-функция* – небольшая анонимная функция, которая обычно создается для того, чтобы потом передавать ее другим функциям в качестве аргумента. Лямбда-функции используются постоянно и повсеместно, поэтому разработчики D сделали все возможное, чтобы избавить программиста от синтаксической нагрузки, прежде неизбежной при определении таких функций: типы параметров и возвращаемый тип выясняются из контекста. Это действительно имеет смысл, потому что тело лямбда-функции по определению там, где нужно автору, читателю и компилятору, то есть здесь нет места разночтениям и принципы модульности не нарушаются.

В связи с лямбда-функцией, определенной в этом примере, стоит упомянуть еще одну деталь. Лямбда-функция осуществляет доступ к переменной freqs, локальной переменной функции main, а значит, лямбда-функция не является ни глобальной, ни статической. Это больше напоминает подход Лиспа, а не C, и позволяет работать с очень мощными лямбда-конструкциями. И хотя обычно за такое преимущество приходится платить неявными вызовами функций во время исполнения программы, D гарантирует отсутствие таких вызовов (и, следовательно, ничем не ограниченные возможности реализации инлайнинга).

Вывод измененной программы:

```
929  the
680  and
625  of
608  to
523  I
453  a
444  my
382  in
361  you
```

```
358  Ham.
```

Что и ожидалось: самые часто употребляемые слова набрали больше всего «очков». Настораживает лишь «Ham»[1]. Это слово в пьесе вовсе не отражает кулинарные предпочтения героев. «Ham» – всего лишь сокращение от «Hamlet» (Гамлет), которым помечена каждая из его реплик. Явно у него был повод высказаться 358 раз – больше, чем любой другой герой пьесы. Далее по списку следует король – всего 116 реплик, меньше трети сказанного Гамлетом. А Офелия с ее 58 репликами – просто молчунья.

1.5. Основные структуры данных

Раз уж мы взялись за «Гамлета», проанализируем этот текст чуть глубже. Например, соберем кое-какую информацию о главных героях: сколько всего слов было произнесено каждым персонажем и насколько богат его (ее) словарный запас. Для этого с каждым действующим лицом понадобится связать несколько фактов. Чтобы сосредоточить эту информацию в одном месте, определим такую структуру данных:

```
struct PersonaData {
   uint totalWordsSpoken;
   uint[string] wordCount;
}
```

В языке D понятия структуры (`struct`) и классы (`class`) четко разделены. С точки зрения удобства они во многом схожи, но устанавливают разные правила: структуры – это типы значений, а классы были задуманы для реализации динамического полиморфизма, поэтому экземпляры классов могут быть доступны исключительно по ссылке. Упразднены связанное с этим непонимание, ошибки при копировании экземпляров классов потомков в переменные классов-предков и комментарии а-ля `// Нет! НЕ наследуй!`. Разрабатывая тип, вы с самого начала должны решить, будет ли это мономорфный тип-значение или полиморфная ссылка. Общеизвестно, что C++ разрешает определять типы, принадлежность которых к тому или иному разряду неочевидна, но эти типы редко используются и чреваты ошибками. В целом достаточно оснований сознательно отказаться от них.

В нашем случае требуется просто собрать немного данных, и мы не планируем использовать полиморфные типы, поэтому тип `struct` – хороший выбор. Теперь определим ассоциативный массив, отображающий имена персонажей на дополнительную информацию о них (значения типа `PersonaData`):

```
PersonaData[string] info;
```

[1] Ham (англ.) – ветчина. – *Прим. пер.*

Все, что нам требуется, – это правильно заполнить info данными из hamlet.txt. Придется немного потрудиться: реплика героя может простираться на несколько строк, и нам понадобится простая обработка, сцепляющая эти физические строки в одну логическую. Чтобы понять, как это сделать, обратимся к небольшому фрагменту файла hamlet.txt, познаково представленному ниже (предшествующие тексту пробелы для наглядности отображаются видимыми знаками):

```
␣␣Pol. Marry, I will teach you! Think yourself a baby
␣␣␣␣That you have ta'en these tenders for true pay,
␣␣␣␣Which are not sterling. Tender yourself more dearly,
␣␣␣␣Or (not to crack the wind of the poor phrase,
␣␣␣␣Running it thus) you'll tender me a fool.
␣␣Oph. My lord, he hath importun'd me with love
␣␣␣␣In honourable fashion.
␣␣Pol. Ay, fashion you may call it. Go to, go to!
```

До сих пор гадают, не было ли истинной причиной гибели Полония злоупотребление инструкцией goto. Но нас больше интересует другое: заметим, что реплике каждого персонажа предшествуют ровно два пробела, за ними следует имя, после которого стоит точка, потом пробел, за которым наконец-то начинается само высказывание. Если логическая строка занимает несколько физических строк, то каждая следующая строка всегда начинается с четырех пробелов. Можно осуществить простое сопоставление шаблону, воспользовавшись регулярными выражениями (для работы с которыми предназначен модуль std.regex), но мы хотим научиться работать с массивами, поэтому выполним сопоставление «вручную». Призовем на помощь лишь логическую функцию a.startsWith(b), определенную в модуле std.algorithm, которая сообщает, начинается ли a с b.

Управляющая функция main читает входную последовательность физических строк, сцепляет их в логические строки (игнорируя все, что не подходит под наш шаблон), передает полученные полные реплики в функцию-накопитель и в конце печатает требуемую информацию:

```
import std.algorithm, std.conv, std.ctype, std.regex,
   std.range, std.stdio, std.string;

struct PersonaData {
   uint totalWordsSpoken;
   uint[string] wordCount;
}

void main() {
   // Накапливает информацию о главных героях
   PersonaData[string] info;
   // Заполнить info
   string currentParagraph;
   foreach (line; stdin.byLine()) {
      if (line.startsWith("  ")
```

```
                && line.length > 4
                && isalpha(line[4])) {
            // Персонаж продолжает высказывание
            currentParagraph ~= line[3 .. $];
        } else if (line.startsWith(" ")
                && line.length > 2
                && isalpha(line[2])) {
            // Персонаж только что начал говорить
            addParagraph(currentParagraph, info);
            currentParagraph = to!string(line[2    $]);
        }
    }
    // Закончили, теперь напечатаем собранную информацию
    printResults(info);
}
```

Зная, как работают массивы, мы без труда читаем этот код, за исключением конструкции `to!string(line[2 .. $])`. Зачем она нужна и что будет, если о ней забыть?

Цикл `foreach`, последовательно считывая из стандартного потока ввода строки текста, размещает их в переменной `line`. Поскольку не имеет смысла выделять память под новый буфер при чтении следующей строки, в каждой итерации `byLine` заново использует место, выделенное для `line`. Тип самой переменной `line` — `char[]`, массив знаков.

Если вы всего лишь, считав, «обследуете» каждую строчку, а потом забываете о ней, в любом случае (как с `to!string(line[2 .. $])`, так и без нее) все будет работать гладко. Но если вы желаете создать код, который будет где-то накапливать содержание читаемых строк, лучше позаботиться о том, чтобы он их действительно копировал. Очевидно, было задумано реально хранить текст в переменной `currentParagraph`, а не использовать ее как временное пристанище, так что необходимо получать дубликаты; отсюда и присутствие конструкции `to!string`, которая преобразует любое выражение в строку. Переменные типа `string` неизменяемы, а `to` гарантирует приведение к этому типу созданием дубликата.

Если забыть написать `to!string` и впоследствии код все же скомпилируется, в результате получится бессмыслица, и ошибку будет довольно-таки сложно обнаружить. Очень неприятно отлаживать программу, одна часть которой изменяет данные, находящиеся в другой части программы, потому что это уже не локальные изменения (трудно представить, сколько вызовов `to` можно забыть при написании большой программы). К счастью, это не причина для беспокойства: типы переменных `line` и `currentParagraph` соответствуют роли этих переменных в программе. Переменная `line` имеет тип `char[]`, представляющий собой массив знаков, которые можно перезаписывать в любой момент; переменная `currentParagraph` имеет тип `string` — массив знаков, которые нельзя изменять по отдельности. (Для самых любопытных: полное имя типа `string` — `immutable(char)[]`, что дословно означает «непрерывный диапазон неизменяемых знаков».

Мы вернемся к разговору о строках в главе 4.) Эти переменные не могут ссылаться на одну и ту же область памяти, поскольку `line` нарушает обязательство `currentParagraph` не изменять знаки по отдельности. Поэтому компилятор отказывает в компиляции ошибочного кода и требует копию, которую вы и предоставляете благодаря преобразованию в строку с помощью конструкции `to!string`. И все счастливы.

С другой стороны, если постоянно копировать строковые значения, то нет необходимости дублировать данные на нижнем уровне их представления – переменные просто могут ссылаться на одну и ту же область памяти, которая наверняка не будет перезаписана. Это делает копирование переменных типа `string` безопасным и эффективным одновременно. Но это еще не все плюсы. Строки можно без проблем разделять между потоками, потому что данные типа `string` неизменяемы, так что возможность конфликта при обращении к памяти попросту отсутствует. Неизменяемость – это действительно здорово. С другой стороны, если вам потребуется интенсивно изменять знаки по отдельности, возможно, вы предпочтете использовать тип `char[]`, хотя бы временно.

Структура `PersonData` в том виде, в каком она задана выше, очень проста. Однако в общем случае структуры могут определять не только данные, но и другие сущности, такие как частные (приватные, закрытые) разделы (обозначаются ключевым словом `private`), функции-члены, тесты модулей, операторы, конструкторы и деструкторы. По умолчанию любой элемент структуры инициализируется значением по умолчанию (ноль для целых чисел, NaN для чисел с плавающей запятой[1] и `null` для массивов и других типов, доступ к которым не осуществляется напрямую. А теперь реализуем функцию `addParagraph`, которая разбивает строку текста на слова и распределяет их по ассоциативному массиву.

Строка, которую обрабатывает `main`, имеет вид: `"Ham. To be, or not to be-that is the question."` Для того чтобы отделить имя персонажа от слов, которые он произносит, нам требуется найти первый разделитель `" "` Для этого используем функцию `find`. Выражение `haystack.find(needle)` возвращает правую часть `haystack`, начинающуюся с первого вхождения `needle`. (Если `needle` в `haystack` отсутствует, то вызов `find` с такими аргументами вернет пустую строку.) Пока мы формируем словарь, не мешает немного прибраться. Во-первых, нужно преобразовать фразу к нижнему регистру, чтобы слово с заглавной и со строчной буквы воспринималось как одна и та же словарная единица. Об этом легко позаботиться с помощью вызова функции `tolower`. Второе, что необходимо сделать, – удалить мощный источник шума – знаки пунктуации, которые превращают, к примеру, «him.» и «him» в разные слова. Для того чтобы очистить словарь, достаточно передать функции `split` единственный дополнительный параметр. Имеется в виду регулярное выраже-

[1] NaN (Not a Number, нечисло) – хорошее начальное значение по умолчанию для чисел с плавающей запятой. К сожалению, для целых чисел не существует эквивалентного начального значения.

ние, которое уничтожит всю «шелуху»: regex("[\t,.;:?]+"). Получив такой аргумент, функция split сочтет любую последовательность знаков, упомянутых между [и], одним из разделителей слов. Теперь мы готовы, как говорится, приносить большую пользу с помощью всего лишь маленького кусочка кода:

```
void addParagraph(string line, ref PersonaData[string] info) {
   // Выделить имя персонажа и его реплику
   line = strip(line);
   auto sentence = std.algorithm.find(line, "  ");
   if (sentence.empty) {
      return;
   }
   auto persona = line[0 .. $ - sentence.length];
   sentence = tolower(strip(sentence[2    $]));
   // Выделить произнесенные слова
   auto words = split(sentence, regex("[ \t, ;:?]+"));
   // Insert or update information
   if (!(persona in info)) {
      // Первая реплика персонажа
      info[persona] = PersonaData();
   }
   info[persona].totalWordsSpoken += words.length;
   foreach (word; words) ++info[persona].wordCount[word];
}
```

Функция addParagraph отвечает за обновление ассоциативного массива. В случае если персонаж еще не высказывался, код вставляет «пустой» объект типа PersonaData, инициализированный значениями по умолчанию. Поскольку значение по умолчанию для типа uint – ноль, а созданный в соответствии с правилами по умолчанию ассоциативный массив пуст, только что вставленный слот готов к приему осмысленной информации.

Наконец, для того чтобы напечатать краткую сводку по каждому персонажу, реализуем функцию printResults:

```
void printResults(PersonaData[string] info) {
   foreach (persona, data; info) {
      writefln("%20s %6u %6u"  persona, data.totalWordsSpoken,
      data.wordCount.length);
   }
}
```

Готовы к тест-драйву? Тогда сохраните и запустите!

```
       Queen  1104   500
         Ros   738   338
         For    55    45
        Fort    74    61
  Gentlemen     4     3
       Other   105    75
```

```
          Guil   349  176
           Mar   423  231
          Capt    92   66
          Lord    70   49
          Both    44   24
           Oph   998  401
         Ghost   683  350
           All    20   17
        Player    16   14
          Laer  1507  606
           Pol  2626  870
        Priest    92   66
           Hor  2129  763
          King  4153 1251
 Cor., Volt     11   11
  Both [Mar      8    8
           Osr   379  179
          Mess   110   79
        Sailor    42   36
       Servant    11   10
 Ambassador      41   34
          Fran    64   47
         Clown   665  298
          Gent   101   77
           Ham 11901 2822
           Ber   220  135
          Volt   150  112
           Rey    80   37
```

Тут есть чем позабавиться. Как и ожидалось, наш дружок «Ham» с большим отрывом выигрывает у всех остальных, получив львиную долю слов. Довольно интересна роль Вольтиманда («Volt»): он немногословен, но при скромном количестве реплик виртуозно демонстрирует солидный словарный запас. Еще любопытнее в этом плане роль матроса («Sailor»), который вообще почти не повторяется. Также сравните красноречивую королеву («Queen») с Офелией («Oph»): королева произносит всего на 10% слов больше, чем Офелия, но ее лексикон богаче как минимум на 25%.

В выводе есть немного шума (например, `"Both [Mar"`), который прилежный программист легко устранит и который вряд ли статистически влияет на то, что действительно представляет для нас интерес. Тем не менее исправление последних огрехов – поучительное (и рекомендуемое) упражнение.

1.6. Интерфейсы и классы

Объектно-ориентированные средства важны для больших проектов; так что, знакомя вас с ними на примере маленьких программ, я рискую выставить себя недоумком. Прибавьте к этому большое желание избе-

жать заезженных примеров с животными и работниками, и сложится довольно неприятная картина. Да, забыл еще кое-что: в маленьких примерах обычно не видны проблемы создания полиморфных объектов, а это очень важно. Что делать бедному автору! К счастью, реальный мир снабдил меня полезным примером в виде относительно небольшой задачи, которая в то же время не имеет удовлетворительного процедурного решения. Обсуждаемый ниже код – это переработка небольшого полезного скрипта на языке awk, который вышел далеко за рамки задуманного. Мы вместе пройдем путь до объектно-ориентированного решения – одновременно компактного, полного и изящного.

Как насчет небольшой программы, собирающей статистику (в связи с этим назовем ее `stats`)? Пускай ее интерфейс будет простым: имена статистических функций, используемых для вычислений, передаются в `stats` как параметры командной строки, последовательность чисел для анализа поступает в стандартный поток ввода в виде списка (разделитель – пробел), статистические результаты печатаются один за другим по одному на строке. Вот пример работы программы:

```
$ echo 3 5 1.3 4 10 4.5 1 5 | stats Min Max Average
1
10
4.225
$ _
```

Написанный на скорую руку «непричесанный» скрипт без проблем решит эту задачу. Но в данном случае при увеличении количества статистических функций «лохматость» кода уничтожит преимущества от быстроты его создания. Так что поищем решение получше. Для начала остановимся на простейших статистических функциях: получение минимума, максимума и среднего арифметического. Нащупав легко расширяемый вариант кода, мы получим простор для неограниченной реализации более сложных статистических функций.

Простейший подход к решению задачи – в цикле пройтись по входным данным и вычислить всю необходимую статистику. Но выбрать такой путь – значит отказаться от идеи масштабируемости программы. Ведь всякий раз, когда нам потребуется добавить новую статистическую функцию, придется подвергать готовый код хирургическому вмешательству. Если мы хотим выполнять только те вычисления, о которых попросили в командной строке, необходимы серьезные изменения. В идеале мы должны заключить все статистические функции в последовательные куски кода. Таким образом, мы расширяем функциональность программы, просто добавляя новый код; это принцип открытости/закрытости [39] во всей красе.

При таком подходе необходимо выяснить, что общего у всех (или хотя бы у большинства) статистических функций. Ведь наша цель – обращаться ко всем функциям из одной точки программы, причем унифицированно. Для начала отметим, что `Min` и `Max` отбирают аргументы из входной

последовательности по одному, а результат будет готов, как только закончится ввод. Конечный результат – одно-единственное число. Также функция Average по окончании чтения всех своих аргументов должна выполнить завершающий шаг (разделить накопившуюся сумму на число слагаемых). Кроме того, у каждого алгоритма есть собственное состояние. Если разные вычисления должны предоставлять одинаковый интерфейс для работы с ними и при этом «запоминать» свое состояние, разумный шаг – сделать их объектами и определить формальный интерфейс для управления всеми этими объектами и каждым из них в отдельности.

```
interface Stat {
   void accumulate(double x);
   void postprocess();
   double result();
}
```

Интерфейс определяет требуемое поведение в виде набора функций. Разумеется, тот, кто замахнется на реализацию интерфейса, должен будет определить все функции в том виде, в каком они заявлены. Раз уж мы заговорили о реализации, давайте посмотрим, как можно определить класс Min, так чтобы он повиновался указаниям железной руки интерфейса Stat.

```
class Min : Stat {
   private double min = double.max;
   void accumulate(double x) {
      if (x < min) {
         min = x;
      }
   }
   void postprocess() {} // Ничего не делать
   double result() {
      return min;
   }
}
```

Min – это *класс*, пользовательский тип, привносящий в D преимущества ООП. С помощью синтаксиса class Min: Stat класс Min во всеуслышание объявляет, что он реализует интерфейс Stat. И Min действительно определяет все три функции, продиктованные волей Stat, в точности с теми же аргументами и возвращаемыми типами (иначе компилятор не дал бы Min просто так проскочить). Min содержит всего лишь один закрытый элемент (тот, что помечен директивой private) – переменную min (наименьшее из прочитанных значений) и обновляет ее внутри функции accumulate. Начальное значение Min – *самое большое* число (которое можно представить типом double), так что первое же число из входной последовательности заместит его.

Перед тем как определить другие статистические функции, реализуем основной алгоритм нашей программы stats, предусматривающий чте-

ние параметров командной строки, создание соответствующих объектов, производящих вычисления (таких как экземпляр класса Min, когда через консоль передан аргумент *Min*), и манипулирование ими с помощью интерфейса Stat.

```
import std.contracts, std.stdio;

void main(string[] args) {
   Stat[] stats;
   foreach (arg; args[1    $]) {
      auto newStat = cast(Stat) Object.factory("stats." ~ arg);
      enforce(newStat, "Invalid statistics function: " ~ arg);
      stats ~= newStat;
   }
   for (double x; stdin. readf(" %s "  &x) == 1; ) {
      foreach (s; stats) {
         s.accumulate(x);
      }
   }
   foreach (s; stats) {
      s.postprocess();
      writeln(s.result());
   }
}
```

Эта небольшая программа творит чудеса. Для начала список параметров main отличается от того, что мы видели до сих пор: на этот раз в функцию передается массив строк. Средства библиотеки времени исполнения D инициализируют этот массив параметрами, переданными компилятору из командной строки вместе с именем скрипта для запуска. Первый цикл инициализирует массив stats исходя из значений массива args. Учитывая, что в D (как и в других языках) первый аргумент – это имя самой программы, мы пропускаем первую позицию: нас интересует срез args[1 .. $]. Теперь разберемся с командой

```
auto newStat = cast(Stat) Object.factory("stats." ~ arg);
```

Тут много непонятного, но, как говорят в ситкомах, я все могу объяснить. Во-первых, здесь знак ~ служит бинарным оператором, то есть осуществляет конкатенацию строк. Поэтому если аргумент командной строки – Min, то результат конкатенации – строка "stats.Min", которая и будет передана функции Object.factory. Object – предок всех классов, создаваемых в программах на D. Он определяет статический метод factory, который принимает строку, ищет соответствующий тип в небольшой базе данных (которая строится во время компиляции), магическим образом создает объект типа, указанного в переданной строке, и возвращает его. Если запрошенный класс отсутствует в упомянутой базе данных, Object.factory возвращает null. Чтобы этого не произошло, достаточно определить класс Min где-нибудь в том же файле, что и вызов Object.factory. Возможность создавать объект по имени его типа – это важное средство, востребованное во множестве полезных при-

ложений. На самом деле, оно настолько важно, что является «сердцем» некоторых языков с динамической типизацией. Языки со статической типизацией (такие как D и Java) вынуждены полагаться на средства своих библиотек времени исполнения или предоставлять программисту самостоятельно изобретать механизмы регистрации и распознавания типов.

Почему `stats.Min`, а не просто `Min`? D серьезно относится к принципу модульности, поэтому в этом языке отсутствует глобальное пространство имен, где кто угодно может складировать что угодно. Каждый символ обитает в рамках модуля со своим именем, и по умолчанию имя модуля совпадает с именем его исходного файла без расширения. Таким образом, при условии что наш файл назван `stats.d`, D полагает, что всякое имя, определенное в этом файле, принадлежит модулю `stats`.

Осталась последняя загвоздка. Статический тип только что полученного объекта типа `Min` на самом деле не `Min`. Это звучит странно, но легко объясняется тем, что, вызвав `Object.factory("что угодно")`, вы можете создать *любой* объект, поэтому возвращаемый тип должен быть неким общим знаменателем для всех возможных объектных типов – и это `Object`. Для того чтобы получить ссылку, соответствующую типу объекта, который вы задумали, необходимо преобразовать объект, возвращенный `Object.factory`, в объект типа `State`. Эта операция называется *приведением типов* (*type casting*). В языке D выражение `cast(T) expr` приводит выражение `expr` к типу `T`. Операции приведения типов, в которых участвуют классы или интерфейсы, всегда проверяются, поэтому код надежно защищен от дураков.

Оглянувшись назад, мы заметим, что львиная доля того, что делает скрипт, выполняется в первых пяти его строках. Эта самая сложная часть, которая полностью определяет весь остальной код. Второй цикл читает по одному числу за раз (об этом заботится функция `readf`) и вызывает `accumulate` для всех объектов, собирающих статистику. Функция `readf` возвращает число объектов, успешно прочитанных согласно заданной строке формата. В нашем случае формат задан в виде строки `" %s "`, что означает «один элемент, окруженный любым количеством пробелов». (Тип элемента определяется типом считанного элемента, в нашем случае `x` принимает значение типа `double`.) Последнее, что делает программа, – выводит результаты вычислений на печать.

1.6.1. Больше статистики. Наследование

Реализация `Max` так же тривиальна, как и реализация `Min`; за исключением небольших изменений в `accumulate`, эти классы ничем не отличаются друг от друга[1]. Даже если новое задание до боли напоминает предыдущее, в голову должна приходить мысль «интересно», а не «о боже,

[1] Это не совсем так. Переменная-аккумулятор должна быть инициализирована значением `double.max` и соответственно переименована. – *Прим. науч. ред.*

какая скука». Рутинные задачи – это возможность для повторного использования, и «правильные» языки, способные лучше эксплуатировать различные преимущества подобия, по некоторой абстрактной шкале качества должны оцениваться выше. Нам придется выяснить, что именно общего у функций Min и Max (и, в идеале, у прочих статистических функций). Присмотревшись к ним, можно заметить, что обе принадлежат к разряду статистических функций, результат которых вычисляется шаг за шагом и может быть вычислен всего по одному числу. Назовем такую категорию статистических функций *пошаговыми функциями*.

```
class IncrementalStat : Stat {
   protected double _result;
   abstract void accumulate(double x);
   void postprocess() {}
   double result() {
      return _result;
   }
}
```

Абстрактный класс можно воспринимать как частичное обязательство: он реализует некоторые методы, но не все, так что «самостоятельно» такой код работать не может. Материализуется абстрактный класс тогда, когда от него наследуют и в теле потомков завершают реализацию. Класс IncrementalStat обслуживает повторяющийся код классов, реализующих интерфейс Stat, но оставляет реализацию метода accumulate своим потомкам. Вот как выглядит новая версия класса Min:

```
class Min : IncrementalStat {
   this() {
      _result = double.max;
   }
   void accumulate(double x) {
      if (x < _result) {
         _result = x;
      }
   }
}
```

Кроме того, в классе Min определен конструктор в виде специальной функции this(), необходимый для корректной инициализации результата. Даже несмотря на добавление конструктора, полученный код значительно улучшил ситуацию относительно исходного положения дел, особенно с учетом того факта, что множество других статистических функций также соответствуют этому шаблону (например, сумма, дисперсия, среднее арифметическое, стандартное отклонение). Посмотрим на реализацию функции получения среднего арифметического, поскольку это прекрасный повод представить еще пару концепций:

```
class Average : IncrementalStat {
   private uint items = 0;
```

```
    this() {
        _result = 0;
    }
    void accumulate(double x) {
        _result += x;
        ++items;
    }
    override void postprocess() {
        if (items) {
            _result /= items;
        }
    }
}
```

Начнем с того, что в Average вводится еще одно поле, items, которое инициализируется нулем с помощью синтаксиса items = 0 (только для того, чтобы показать, как надо инициализировать переменные, но, как отмечалось выше, целые числа и так инициализируются нулем по умолчанию). Второе, что необходимо отметить: Average определяет конструктор, который присваивает переменной _result ноль. Так сделано, потому что, в отличие от минимума или максимума, при отсутствии аргументов среднее арифметическое считается равным нулю. И хотя может показаться, что инициализировать _result значением NaN только для того, чтобы тут же записать в эту переменную ноль, – бессмысленное действие, уход от так называемого «мертвого присваивания» представляет собой легкую добычу для любого оптимизатора. Наконец, Average переопределяет метод postprocess, несмотря на то что в классе IncrementalStat он уже определен. В языке D по умолчанию можно переопределить (унаследовать и заново определить) методы любого класса, но надо обязательно добавлять директиву override, чтобы избежать всевозможных несчастных случаев (таких как неудача переопределения в связи с какой-нибудь опечаткой или изменением в базовом типе, либо переопределение чего-нибудь по ошибке). Если вы поставите перед методом класса ключевое слово final, то запретите классам-потомкам переопределять эту функцию (что эффективно останавливает механизм динамического поиска методов по дереву классов).

1.7. Значения против ссылок

Проведем небольшой эксперимент:

```
import std.stdio;

struct MyStruct {
    int data;
}
class MyClass {
    int data;
}
```

```
void main() {
   // Играем с объектом типа MyStruct
   MyStruct s1;
   MyStruct s2 = s1;
   ++s2.data;
   writeln(s1.data); // Печатает 0
   // Играем с объектом типа MyClass
   MyClass c1 = new MyClass;
   MyClass c2 = c1;
   ++c2.data;
   writeln(c1.data); // Печатает 1
}
```

Похоже, игры с объектом типа `MyStruct` сильно отличаются от игр с объектом типа `MyObject`. И в том и в другом случае мы создаем переменную, которую затем копируем в другую переменную, после чего изменяем копию (вспомните, что ++ – это унарный оператор, прибавляющий единицу к своему аргументу). Этот эксперимент показывает, что после копирования `c1` и `c2` ссылаются на одну и ту же область памяти с информацией, а `s1` и `s2`, напротив, «живут врозь».

Поведение `MyStruct` свидетельствует о том, что этот объект подчиняется *семантике значений*: каждая переменная ссылается на собственное единственное значение, и присваивание одной переменной другой означает, что значение одной переменной реально копируется в значение другой переменной. Исходное значение, по образу и подобию которого изменяли вторую переменную, остается нетронутым, и обе переменные далее продолжают развиваться независимо друг от друга. Поведение `MyClass` говорит, что объект этого типа подчиняется *ссылочной семантике*: значения создаются явно (в нашем случае с помощью вызова `new MyClass`), и присваивание одного экземпляра класса другому означает лишь то, что обе переменные будут ссылаться на одно и то же значение в памяти.

Со значениями легко работать, о них просто рассуждать, и они позволяют производить эффективные вычисления с переменными небольшого размера. С другой стороны, нетривиальные программы сложно реализовать, не обладая средствами доступа к переменным без их копирования. Отсутствие возможности работать со ссылками препятствует, например, работе с типами, ссылающимися на себя же (списки или деревья), или структурами, ссылающимися друг на друга (такими как дочернее окно, знающее о своем родительском окне). Любой уважающий себя язык реализует работу со ссылками в том или ином виде; спорят только о необходимых умолчаниях. В C в общем случае переменные трактуются как значения, но если пользователь захочет, он будет работать со ссылками с помощью указателей. В дополнение к указателям, C++ определяет ссылочные типы. Любопытно, что чисто функциональные языки могут использовать ссылки или значения, когда сочтут нужным, потому что при написании кода между ними нет разницы. Ведь

чисто функциональные языки запрещают изменения, поэтому невозможно сказать, когда они порождают копию значения, а когда просто используют ссылку на него – значения «заморожены», поэтому вы не сможете проверить, разделяется ли значение между несколькими переменными, изменив одну из них. Чисто объектно-ориентированные языки, напротив, традиционно поощряют изменения. Для них общий случай – ссылочная семантика. Некоторые такие языки достигают умопомрачительной гибкости, допуская, например, динамическое изменение системных переменных. Наконец, некоторые языки избрали гибридный подход, включая как типы-значения, так и ссылочные типы, с разной долей предпочтения тем или другим.

Язык D систематически реализует гибридный подход. Для определения ссылочных типов используйте классы. Для определения типов-значений или гибридных типов используйте структуры. В главах 6 и 7 соответственно описаны конструкторы этих типов, снабженные средствами для реализации соответствующего подхода. Например, структуры не поддерживают динамическое наследование и полиморфизм (такой как в рассмотренной нами программе `stats`), поскольку такое поведение не согласуется с семантикой значений. Динамический полиморфизм объектов – это характеристика ссылочной семантики, и любая попытка смешать эти два подхода приведет лишь к жутким последствиям. (Например, классическая опасность, подстерегающая программистов на C++, – slicing, неожиданное лишение объекта его полиморфных способностей в результате невнимательного использования этого объекта в качестве значения. В языке D slicing невозможен.)

В завершение хочется сказать, что структуры – пожалуй, наиболее гибкое проектное решение. Определив структуру, вы можете вдохнуть в нее любую семантику. Вы можете сделать так, что значение будет копироваться постоянно, реализовать ленивое копирование, а-ля копирование при записи, или подсчитывать ссылки, или выбрать что-то среднее между этими способами. Вы даже можете определить ссылочную семантику, используя классы или указатели *внутри* своей структуры. С другой стороны, некоторые из этих альтернатив требуют подкованности в техническом плане; использование классов, напротив, подразумевает простоту и унифицированность.

1.8. Итоги

Эта глава – вводная, поэтому какие-то детали отдельных примеров и концепций остались за кадром или были рассмотрены вскользь. При этом опытный программист легко поймет, как можно завершить и усовершенствовать код примеров.

Надеюсь, что-то интересное нашлось для каждого. Кодера-практика, противника любых излишеств, могла порадовать чистота синтаксиса массивов и ассоциативных массивов. Уже эти две концепции сильно

упрощают ежедневное кодирование и полезны как для малых, так и для больших проектов. Поклонник объектно-ориентированного программирования, хорошо знакомый с интерфейсами и классами, мог отметить хорошую масштабируемость языка для крупных проектов. А те, кто хочет писать на D короткие скрипты, увидели, как легко пишутся и запускаются сценарии, манипулирующие файлами.

Как водится, полный рассказ гораздо длиннее. И все же полезно время от времени вернуться к основам и удостовериться, что простые вещи остаются простыми.

2

Основные типы данных. Выражения

Если вы когда-нибудь программировали на C, C++, Java или C#, то с основными типами данных и выражениями D у вас не будет никаких затруднений. Операции со значениями основных типов – неотъемлемая часть решений многих задач программирования. Эти средства языка, в зависимости от ваших предпочтений, могут сильно облегчать либо отравлять вам жизнь. Совершенного подхода не существует; нередко поставленные цели противоречат друг другу, заставляя руководствоваться собственным субъективным мнением. Это, в свою очередь, лишает язык возможности угодить всем до единого. Слишком строгая система обременяет программиста своими запретами: он вынужден бороться с компилятором, чтобы тот принял простейшие выражения. А сделай систему типизации чересчур снисходительной – и не заметишь, как окажешься по ту сторону корректности, эффективности или того и другого вместе.

Система основных типов D творит маленькие чудеса в границах, задаваемых его принадлежностью к семейству статически типизированных компилируемых языков. Определение типа по контексту, распространение интервала значений, всевозможные стратегии перегрузки операторов и тщательно спроектированная сеть автоматических преобразований вместе делают систему типизации D дотошным, но сдержанным помощником, который если и придирается, требуя внимания, то обычно не зря.

Основные типы данных можно распределить по следующим категориям:

- *Тип без значения*: `void`, используется во всех случаях, когда формально требуется указать тип, но никакое осмысленное значение не порождается.
- *Тип null*: `typeof(null)` – тип константы `null`, используется в основном в шаблонах, неявно приводится к указателям, массивам, ассоциативным массивам и объектным типам.

- *Логический (булев) тип*: bool с двумя возможными значениями true и false.
- *Целые типы*: byte, short, int и long, а также их эквиваленты без знака ubyte, ushort, uint и ulong.
- *Вещественные типы с плавающей запятой*: float, double и real.
- *Знаковые типы*: char, wchar и dchar, которые на самом деле содержат числа, предназначенные для кодирования знаков Юникода.

В табл. 2.1 вкратце описаны основные типы данных D с указанием их размеров и начальных значений по умолчанию. В языке D переменная инициализируется автоматически, если вы просто определили ее, не указав начального значения. Значение по умолчанию доступно для любого типа как <*тип*>.init; например int.init — это ноль.

Таблица 2.1. Основные типы данных D

Тип данных	Описание	Начальное значение по умолчанию
void	Без значения	n/a
typeof(null)	Тип константы null	n/a
bool	Логическое (булево) значение	false
byte	Со знаком, 8 бит	0
ubyte	Без знака, 8 бит	0
short	Со знаком, 16 бит	0
ushort	Без знака, 16 бит	0
int	Со знаком, 32 бита	0
uint	Без знака, 32 бита	0
long	Со знаком, 64 бита	0
ulong	Без знака, 64 бита	0
float	32 бита, с плавающей запятой	float.nan
double	64 бита, с плавающей запятой	double.nan
real	Наибольшее, какое только может позволить аппаратное обеспечение	real.nan
char	Без знака, 8 бит, в UTF-8	0xFF
wchar	Без знака, 16 бит, в UTF-16	0xFFFF
dchar	Без знака, 32 бита, в UTF-32	0x0000FFFF

2.1. Идентификаторы

Идентификатор, или символ – это чувствительная к регистру строка знаков, начинающаяся с буквы или знака подчеркивания, после чего следует любое количество букв, знаков подчеркивания или цифр. Единственное исключение из этого правила: идентификаторы, начинающиеся с двух знаков подчеркивания, зарезервированы под ключевые слова самого D. Идентификаторы, начинающиеся с одного знака подчеркивания, разрешены, и в настоящее время даже принято именовать поля классов таким способом.

Интересная особенность идентификаторов D – их интернациональность: «буква» в определении выше – это не только буква латинского алфавита (от A до Z и от a до z), но и знак из универсального набора[1], определенного в стандарте C99[2] [33, приложение к D].

Например, abc, α5, _, Γ_1, _AbC, Ab9C и _9x – допустимые идентификаторы, а 9abc и __abc – нет.

Если перед идентификатором стоит точка (.какЗдесь), то компилятор ищет его в пространстве имен модуля, а не в текущем лексически близком пространстве имен. Этот префиксный оператор-точка имеет тот же приоритет, что и обычный идентификатор.

2.1.1. Ключевые слова

Приведенные в табл. 2.2 идентификаторы – это ключевые слова, зарезервированные языком для специального использования. Пользовательский код не может переопределять их ни при каких условиях.

Таблица 2.2. Ключевые слова языка D

abstract	else	macro	switch
alias	enum	mixin	synchronized
align	export	module	
asm	extern		template
assert		new	this
auto	false	nothrow	throw
	final	null	true
body	finally		try
bool	float	out	typeid
break	for	override	typeof

[1] Впрочем, использование нелатинских букв является дурным тоном. – *Прим. науч. ред.*

[2] C99 – обновленная спецификация C, в том числе добавляющая поддержку знаков Юникода. – *Прим. пер.*

byte	foreach		
	function	package	ubyte
case		pragma	uint
cast	goto	private	ulong
catch		protected	union
char	ifIf	public	unittest
class	immutable	pure	ushort
const	import		
continue	in	real	version
	inout	ref	void
dchar	int	return	
debug	interface		wchar
default	invariant	scope	while
delegate	isIs	short	with
deprecated		static	
do	long	struct	
double	lazy	super	

Некоторые из ключевых слов распознаются как первичные выражения. Например, ключевое слово `this` внутри определения метода означает текущий объект, а ключевое слово `super` как статически, так и динамически заставляет компилятор обратиться к классу-родителю текущего класса (см. главу 6). Идентификатор `$` разрешен только внутри индексного выражения или выражения получения среза и обозначает длину индексируемого массива. Идентификатор `null` обозначает пустой объект, массив или указатель.

Первичное выражение `typeid(T)` возвращает информацию о типе `T` (за дополнительной информацией обращайтесь к документации для вашей реализации компилятора).

2.2. Литералы

2.2.1. Логические литералы

Логические (булевы) литералы – это `true` («истина») и `false` («ложь»).

2.2.2. Целые литералы

D работает с десятичными, восьмеричными[1], шестнадцатеричными и двоичными целыми литералами. Десятичная константа – это последовательность цифр, возможно, с суффиксом `L`, `U`, `u`, `LU`, `Lu`, `UL` или `ul`. Вывод о типе десятичного литерала делается исходя из следующих правил:

- *нет суффикса*: если значение «помещается» в `int`, то `int`, иначе `long`;
- *только U/u*: если значение «помещается» в `uint`, то `uint`, иначе `ulong`.
- *только L*: тип константы – `long`.
- *U/u и L совместно*: тип константы – `ulong`.

Например:

```
auto
  a = 42,           // a имеет тип int
  b = 42u,          // b имеет тип uint
  c = 42UL,         // c имеет тип ulong
  d = 4000000000,   // long; в int не поместится
  e = 4000000000u,  // uint; в uint не поместится
  f = 5000000000u;  // ulong; в uint не поместится
```

Вы можете свободно вставлять в числа знаки подчеркивания (только не ставьте их в начало, иначе вы на самом деле создадите идентификатор). Знаки подчеркивания помогают сделать большое число более наглядным:

```
auto targetSalary = 15_000_000;
```

Чтобы написать шестнадцатеричное число, используйте префикс `0x` или `0X`, за которым следует последовательность знаков `0-9`, `a-f`, `A-F` или `_`. Двоичный литерал создается с помощью префикса `0b` или `0B`, за которым идет последовательность из `0`, `1` и тех же знаков подчеркивания. Как и у десятичных чисел, у всех этих литералов может быть суффикс. Правила, с помощью которых их типы определяются по контексту, идентичны правилам для десятичных чисел.

Рисунок 2.1, заменяющий 1024 слова, кратко и точно определяет синтаксис целых литералов. Правила интерпретации автомата таковы: 1) каждое ребро «поглощает» знаки, соответствующие его ребру, 2) автомат пытается «расходовать» как можно больше знаков из входной последовательности[2]. Достижение конечного состояния (двойной кружок) означает, что число успешно распознано.

---

[1] Сам язык не поддерживает восьмеричные литералы, но поскольку они присутствуют в некоторых C-подобных языках, в стандартную библиотеку был добавлен соответствующий шаблон. Теперь запись `std.conv.octal!777` аналогична записи `0777` в C. – *Прим. науч. ред.*

[2] Для тех, кто готов воспринимать теорию: автоматы на рис. 2.1 и 2.2 – это детерминированные конечные автоматы (ДКА).

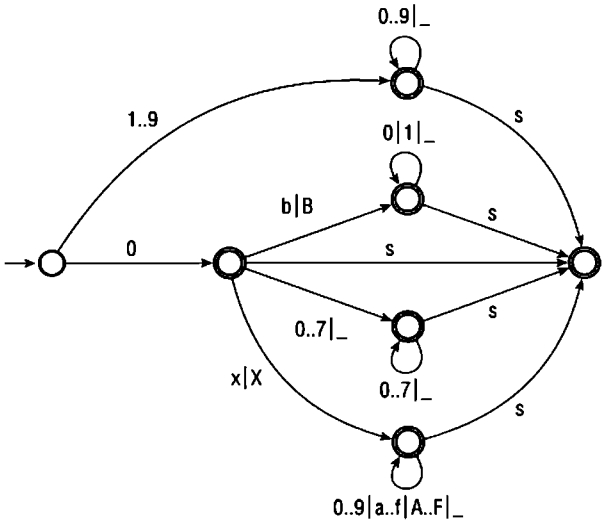

***Рис. 2.1.** Распознавание целых литералов в языке D. Автомат пытается сделать ряд последовательных шагов (поглощая знаки, соответствующие данному ребру), пока не остановится. Останов в конечном состоянии (двойной кружок) означает, что число успешно распознано.*
s обозначает суффикс вида U|u|L|UL|uL|Lu|LU

2.2.3. Литералы с плавающей запятой

Литералы с плавающей запятой могут быть десятичными и шестнадцатеричными. Десятичные литералы с плавающей запятой легко определить по аналогии с только что определенными десятичными целыми числами: десятичный литерал с плавающей запятой состоит из десятичного литерала, который также может содержать точку[1] в любой позиции, за ней могут следовать показатель степени (характеристика) и суффикс. Показатель степени[2] – это то, что обозначается как e, E, e+, E+, e- или E-, после чего следует целый десятичный литерал без знака[3]. В качестве суффикса может выступать f, F или L. Разумеется, хотя бы что-то одно из e/E и f/F должно присутствовать, иначе если в числе нет

[1] В России в качестве разделителя целой и дробной части чисел с плавающей запятой принята запятая (поэтому и говорят: «числа с плавающей *запятой*»), однако в англоговорящих странах для этого служит точка, поэтому в языках программирования (обычно основанных на английском – международном языке информатики) разделителем является точка. – *Прим. пер.*

[2] Показатель степени 10 по-английски – exponent, поэтому для его обозначения и используется буква e. – *Прим. пер.*

[3] Запись Ep означает «умножить на 10 в степени p», то есть p – это порядок. – *Прим. пер.*

точки, вместо числа с плавающей запятой получим целое. Суффикс f/F заставляет компилятор определить тип литерала как float, а суффикс L – как real. Иначе литералу будет присвоен тип double.

Может показаться, что шестнадцатеричные константы с плавающей запятой – вещь странноватая. Однако, как показывает практика, они очень удобны, если нужно записать число очень *точно*. Внутреннее представление чисел с плавающей запятой характеризуется тем, что числа хранятся в двоичном виде, поэтому запись вещественного числа в десятичном виде повлечет преобразования, невозможные без округлений, поскольку 10 – не степень 2. Шестнадцатеричная форма записи, напротив, позволяет записать число с плавающей запятой точно так, как оно будет представлено. Полный курс по представлению чисел с плавающей запятой выходит за рамки этой книги; отметим лишь, что все реализации D гарантированно используют формат IEEE 754, полную информацию о котором можно найти в Сети (сделайте запрос «формат чисел с плавающей запятой IEEE 754»).

Шестнадцатеричный литерал с плавающей запятой состоит из префикса 0x или 0X, за которым следует строка шестнадцатеричных цифр, содержащая точку в любой позиции. Затем идет обязательный показатель степени[1], который начинается с p, P, p+, P+, p- или P- и заканчивается десятичными (не шестнадцатеричными!) цифрами. Только так называемая мантисса – дробная часть перед показателем степени – выражается шестнадцатеричным числом; сам показатель степени – целое десятичное число. Показатель степени шестнадцатеричной константы с плавающей запятой означает степень 2 (а не 10, как в случае с десятичным представлением). Завершается литерал необязательным суффиксом f, F или L[2]. Рассмотрим несколько подходящих примеров:

```
auto
  a = 1.0,      // a имеет тип double
  b = .345E2f,  // b = 34.5 имеет тип float
  c = 10f,      // c имеет тип float из-за суффикса
  d = 10.       // d имеет тип double
  e = 0x1.fffffffffffffp1023, // наибольшее возможное
                              // значение типа double
  f = 0XFp1F;   // f = 30.0, тип float
```

Рисунок 2.2 без лишних слов описывает литералы с плавающей запятой языка D. Правила интерпретации автомата те же, что и для автомата, иллюстрирующего распознавание целых литералов: переход выполняется по мере чтения знаков литерала с целью прочитать как можно

[1] Степень по-английски – power, поэтому показатель степени 2 обозначается буквой p. – *Прим. пер.*

[2] Да, синтаксис странноватый, но D скопировал его из стандарта C99, чтобы не изобретать свою нотацию с собственными выкрутасами, которых все равно не избежать.

больше. Представление в виде автомата проясняет несколько фактов, которые было бы утомительно описывать, не используя формальный аппарат. Например, 0x.p1 и 0xp1 – вполне приемлемые, хотя и странные формы записи нуля, а конструкции типа 0e1, .e1 и 0x0.0 запрещены.

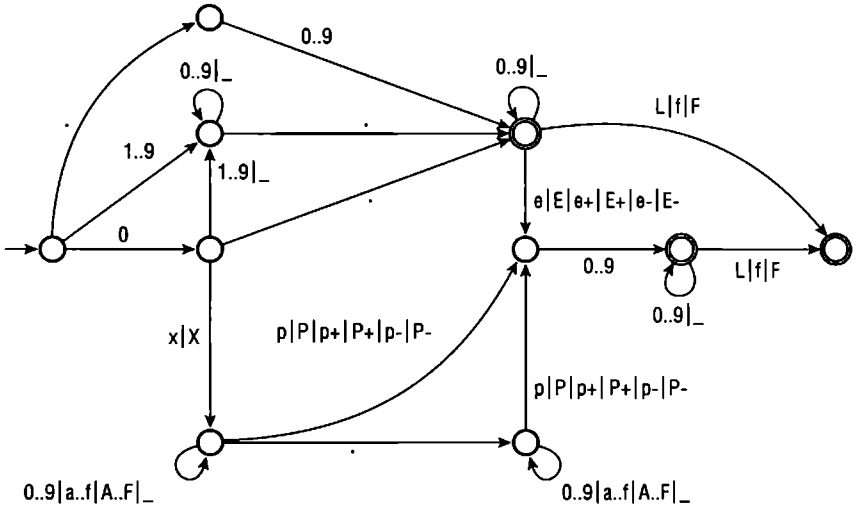

Рис. 2.2. Распознавание литералов с плавающей запятой

2.2.4. Знаковые литералы

Знаковый литерал – это один знак, заключенный в одиночные кавычки, например 'a'. Если в качестве знака выступают сами кавычки, их нужно экранировать с помощью обратной косой черты: '\''. На самом деле в D, как и в других языках, определены несколько разных escape-последовательностей[1] (см. табл. 2.3). В дополнение к стандартному набору управляющих непечатаемых символов D предоставляет следующие возможности записать знаки Юникода: '\u03C9' (знаки \u, за которыми следуют ровно 4 шестнадцатеричные цифры), '\U0000211C' (знаки \U, за которыми следуют ровно 8 шестнадцатеричных цифр) и '\©' (имя, окруженное знаками \& и ;). Первый из этих примеров – знак ω в Юникоде, второй – красивая письменная $\mathscr{R}$, а последний – грозный знак ©. Если вам нужен полный список знаков, которые можно отобразить, поищите в Интернете информацию о таблице знаков Юникода.

[1] Escape-последовательность (от англ. escape – избежать), экранирующая/управляющая последовательность – специальная комбинация знаков, отменяющая стандартную обработку компилятором следующих за ней знаков (они как бы «исключаются из рассмотрения»). – *Прим. пер.*

Таблица 2.3. Экранирующие последовательности в D

Escape-последовательность	Тип	Описание
\"	char	Двойная кавычка (если двусмысленно)
\\	char	Обратная косая черта
\a	char	Звуковой сигнал (Bell, ASCII 7)
\b	char	Backspace (ASCII 8)
\f	char	Смена страницы (ASCII 12)
\n	char	Перевод строки (ASCII 10)
\r	char	Возврат каретки (ASCII 13)
\t	char	Табуляция (ASCII 9)
\v	char	Вертикальная табуляция (ASCII 11)
\<*1-3 восьмеричные цифры*>	char	Знак UTF-8 в восьмеричном представлении (не больше 377_8)
\x<*2 шестнадцатеричные цифры*>	char	Знак UTF-8 в шестнадцатеричном представлении
\u<*4 шестнадцатеричные цифры*>	wchar	Знак UTF-16 в шестнадцатеричном представлении
\U<*8 шестнадцатеричных цифр*>	dchar	Знак UTF-32 в шестнадцатеричном представлении
\&<*имя знака*>;	dchar	Имя знака Юникод

2.2.5. Строковые литералы

Теперь, когда мы знаем, как представляются отдельные знаки, строковые литералы для нас пустяк. D прекрасно справляется с обработкой строк отчасти благодаря своим мощным средствам представления строковых литералов. Как и другие языки, работающие со строками, D различает строки, заключенные в кавычки (внутри которых можно размещать экранированные последовательности из табл. 2.3), и WYSIWYG-строки[1] (которые компилятор распознает «вслепую», не пытаясь обнаружить и расшифровать никакие escape-последовательности). Стиль WYSIWYG очень удобен для представления строк, где иначе пришлось бы использовать множество экранированных знаков; два выдающихся примера – регулярные выражения и пути к файлам в системе Windows.

[1] WYSIWIG – акроним «What You See Is What You Get» (что видишь, то и получишь) – способ представления, при котором данные в процессе редактирования выглядят так же, как и в результате обработки каким-либо инструментом (компилятором, после отображения браузером и т. п.). – *Прим. пер.*

Строки, заключенные в кавычки (quoted strings), – это последовательности знаков в двойных кавычках, "как в этом примере". В таких строках все escape-последовательности из табл. 2.3 являются значимыми. Строки всех видов, расположенные подряд, автоматически подвергаются конкатенации:

```
auto crlf = "\r\n";
auto a = "В этой строке есть \"двойные кавычки\"  а также
перевод строки, даже два" "\n";
```

Текст умышленно перенесен на новую строку после слова также: строковый литерал может содержать знак перевода строки (реальное начало новой строки в исходном коде, а не комбинацию \n), который будет сохранен именно в этом качестве.

2.2.5.1. Строковые литералы: WYSIWYG, с разделителями, строки токенов, шестнадцатеричные и импортированные

WYSIWYG-строка либо начинается с r" и заканчивается на " (r"как здесь"), либо начинается и заканчивается грависом[1] (`как здесь`). В WYSIWYG-строке может встретиться любой знак (кроме соответствующих знаков начала и конца литерала), который хранится так же, как и выглядит. Это означает, что вы не можете представить, например, знак двойной кавычки внутри заключенной в такие же двойные кавычки WYSIWYG-строки. Это не проблема, потому что всегда можно сделать конкатенацию строк, представленных с помощью разных синтаксических правил. Например:

```
auto a = r"Строка с \ и " `"` " внутри.";
```

Из практических соображений можно считать, что двойная кавычка внутри r"такой строки" обозначается последовательностью (`"`), а гравис внутри `такой строки` – последовательностью (`"`). Счастливого подсчета кавычек.

Иногда бывает удобно описать строку, ограниченную с двух сторон какими-то символами. Для этих целей D предоставляет особый вид строковых литералов – литерал с разделителями.

```
auto a = q"[Какая-то строка с "кавычками"  `обратными апострофами` и [квадратными скобками]]";
```

Теперь в a находится строка, содержащая кавычки, обратные апострофы и квадратные скобки, то есть все, что мы видим между q"[и]". И никаких обратных слэшей, нагромождения ненужных кавычек и прочего мусора. Общий формат этого литерала: сначала идет префикс q и двойная кавычка, за которой сразу без пробелов следует знак-разделитель. Оканчивается строка знаком-разделителем и двойной кавычкой, за которой может следовать суффикс, указывающий на тип литерала: c, w

[1] Он же обратный апостроф. – *Прим. науч. ред.*

или d. Допустимы следующие парные разделители: [и], (и), < и >, { и }. Допускаются вложения парных разделителей, то есть внутри пары скобок может быть другая пара скобок, которая становится частью строки. В качестве разделителя можно также использовать любой знак, например:

```
auto a = q"/Просто строка/"; // Эквивалентно строке "Просто строка"
```

При этом строка распознается до первого вхождения ограничивающего знака:

```
auto a = q"/Просто/строка/"; // Ошибка.
auto b = q"[]]";             // Опять ошибка.
auto c = q"[[]]";            // А теперь все нормально,
                             // т. к. разделители [] допускают вложение.
```

Если в качестве разделителя нужно использовать какую-то последовательность знаков, открывающую и закрывающую последовательности следует писать на отдельной строке:

```
auto a = q"EndOfString
Несколько
строк
текста
EndOfString";
```

Кроме того, D предлагает такой вид строкового литерала, как строка токенов. Такой литерал начинается с последовательности q{ и заканчивается на }. При этом текст, расположенный между { и }, должен представлять из себя последовательность токенов языка D и интерпретируется как есть.

```
auto a = q{ foo(q{hello}); }; // Эквивалентно " foo(q{hello}); "
auto b = q{ № };              // Ошибка! "№" - не токен языка
auto a = q{ __EOF__ };        // Ошибка! __EOF__ - не токен, а конец файла
```

Также D определяет еще один вид строковых литералов – шестнадцатеричную строку, то есть строку, состоящую из шестнадцатеричных цифр и пробелов (пробелы игнорируются) между x" и ". Шестнадцатеричные строки могут быть полезны для определения сырых данных; компилятор не пытается интерпретировать содержимое литералов ни как знаки Юникода, ни как-то еще – только как шестнадцатеричные цифры. Пробелы внутри строк игнорируются.

```
auto
a = x"0A"           // То же самое, что "\x0A"
b = x"00 F BCD 32"; // То же самое, что "\x00\xFB\xCD\x32"
```

Если ваш хакерский мозг уже начал прикидывать, как внедрить в программы на D двоичные данные, вы будете счастливы услышать об очень мощном способе определения строки: из файла!

```
auto x = import("resource.bin");
```

Во время компиляции переменная x будет инициализирована непосредственно содержимым файла resource.bin. (Это отличается от действия директивы C #include, поскольку в рассмотренном примере файл включается в качестве данных, а не кода.) По соображениям безопасности допустимы только относительные пути и пути поиска контролируются флагами компилятора. Эталонная реализация dmd использует флаг -J для управления путями поиска.

Строка, возвращаемая функцией import, не проверяется на соответствие кодировке UTF-8. Это сделано намеренно – для реализации возможности включать двоичные данные.

2.2.5.2. Тип строкового литерала

Каков тип строкового литерала? Проведем простой эксперимент:

```
import std.stdio;
void main() {
   writeln(typeid(typeof("Hello, world!")));
}
```

Встроенный оператор typeof возвращает тип выражения, а typeid конвертирует его в печатаемую строку. Наша маленькая программа печатает:

```
immutable(char)[]
```

открывая нам то, что строковые литералы – это *массивы неизменяемых знаков*. На самом деле, тип string, который мы использовали в примерах, – это краткая форма записи (или псевдоним), означающая immutable(char)[]. Рассмотрим подробно все три составляющие типа строковых литералов: неизменяемость, длина и базовый тип данных – знаковый.

Неизменяемость

Строковые литералы живут в неизменяемой области памяти. Это вовсе не говорит о том, что они хранятся на действительно не стираемых кристаллах ЗУ или в защищенной области памяти операционной системы. Это означает, что язык обязуется не перезаписывать память, выделенную под строку. Ключевое слово immutable воплощает это обязательство, запрещая во время компиляции любые операции, которые могли бы модифицировать содержимое неизменяемых данных, помеченных этим ключевым словом:

```
auto a = "Изменить этот текст нельзя";
a[0] = 'X'; // Ошибка! Нельзя модифицировать неизменяемую строку!
```

Ключевое слово immutable – это *квалификатор типа* (квалификаторы обсуждаются в главе 8); его действие распространяется на любой тип, указанный в круглых скобках после него. Если вы напишете immutable(char)[] str, то знаки в строке str нельзя будет изменять по отдельности, однако str можно заставить ссылаться на другую строку:

```
immutable(char)[] str = "One";
str[0] = 'X'; // Ошибка! Нельзя присваивать значения
              // переменным типа immutable(char)!
str = "Two";  // Отлично, присвоим str другую строку
```

С другой стороны, если круглые скобки отсутствуют, квалификатор `immutable` будет относиться ко всему массиву:

```
immutable char[] a = "One";
a[0] = 'X'; // Ошибка!
a = "Two";  // Ошибка!
```

У неизменяемости масса достоинств, а именно: квалификатор `immutable` предоставляет достаточно гарантий, чтобы разрешить неразборчивое совместное использование данных модулями и потоками (см. главу 13). Поскольку знаки строки неизменяемы, никогда не возникает споров, и совместный доступ безопасен и эффективен.

Длина

Очевидно, что длина строкового литерала (13 для `"Hello, world!"`) известна во время компиляции. Поэтому может казаться естественным давать наиболее точное определение каждой строке; например, строка `"Hello, world!"` может быть типизирована как `char[13]`, что означает «массив ровно из 13 знаков». Однако опыт языка Паскаль показал, что статические размеры крайне неудобны. Так что в D тип литерала не включает информацию о его длине. Тем не менее, если вы действительно хотите работать со строкой фиксированного размера, то можете создать такую, явно указав ее длину:

```
immutable(char)[13] a = "Hello, world!";
char[13] b = "Hello, world!";
```

Типы массивов фиксированного размера `T[N]` могут неявно конвертироваться в типы динамических массивов `T[]` для всех типов `T`. В процессе преобразования информация не теряется, так как динамические массивы «помнят» свою длину:

```
import std.stdio;
void main() {
   immutable(char)[3] a = "Hi!";
   immutable(char)[] b = a;
   writeln(a.length, " "  b.length); // Печатает "3 3"
}
```

Базовый тип данных – знаковый

Последнее, но немаловажное, что нужно сказать о строковых литералах, – в качестве их базового типа может выступать `char`, `wchar` или `dchar`[1].

[1] Префиксы `w` и `d` – от англ. wide (широкий) и double (двойной) – *Прим. науч. ред.*

Использовать многословные имена типов необязательно: string, wstring и dstring – удобные псевдонимы для immutable(char)[], immutable(wchar)[] и immutable(dchar)[] соответственно. Если строковый литерал содержит хотя бы один 4-байтный знак типа dchar, то строка принимает тип dstring; иначе, если строка содержит хотя бы один 2-байтный знак типа wchar, то строка принимает тип wstring, иначе строка принимает знакомый тип string. Если ожидается тип, отличный от определяемого по контексту, литерал молча уступит, как в этом примере:

```
wstring x = "Здравствуй, широкий мир!"c;          // UTF-16
dstring y = "Здравствуй, еще более широкий мир!"c; // UTF-32
```

Если вы хотите явно указать тип строки, то можете снабдить строковый литерал суффиксом: c, w или d, которые заставляют тип строкового литерала принять значение string, wstring или dstring соответственно.

2.2.6. Литералы массивов и ассоциативных массивов

Строка – это частный случай массива со своим синтаксисом литералов. А как представить литерал массива другого типа, например int или double? Литерал массива задается заключенным в квадратные скобки списком значений, разделенных запятыми[1]:

```
auto somePrimes = [ 2u, 3, 5, 7, 11, 13 ];
auto someDoubles = [ 1.5, 3, 4.5 ];
```

Размер массива вычисляется по количеству разделенных запятыми элементов списка. В отличие от строковых литералов, литералы массивов изменяемы, так что вы можете изменить их после инициализации:

```
auto constants = [ 2.71, 3.14, 6.023e22 ];
constants[0] = 2.21953167; // "Константа дивана"
auto salutations = [ "привет"  "здравствуйте"  "здорово" ];
salutations[2] = "Да здравствует Цезарь";
```

Обратите внимание: можно присвоить новую строку элементу массива salutations, но нельзя изменить содержимое старой строки, хранящееся в памяти. Этого и следовало ожидать, потому что членство в массиве не отменяет правил работы с типом string.

Тип элементов массива определяется «соглашением» между всеми элементами массива, которое вычисляется с помощью оператора сравнения ?: (см. раздел 2.3.16). Для литерала lit, содержащего больше одного элемента, компилятор вычисляет выражение true ? lit[0] : lit[1]

---

[1] В литерале массива допустима запятая, после которой нет элемента, например [1, 2,] – длина этого массива равна 2, а последняя запятая попросту игнорируется. Это сделано для удобства автоматических генераторов кода: при генерации текста литерала массива они конкатенируют строки вида "очередной_элемент", не обрабатывая отдельно последний элемент, запятая после которого была бы не нужна. – *Прим. науч. ред.*

и сохраняет тип этого выражения как тип L. Затем для каждого i-го элемента lit[i] до последнего элемента в lit компилятор вычисляет тип true ? L.init : lit[i] и снова сохраняет это значение в L. Конечное значение L и есть тип элементов массива.

На самом деле, все гораздо проще, чем кажется, – тип элементов массива устанавливается аналогично Польскому демократическому соглашению[1]: ищется тип, в который можно неявно конвертировать все элементы массива. Например, тип массива [1, 2, 2.2] – double, а тип массива [1, 2, 3u] – uint, так как результатом операции ?: с аргументами int и uint будет uint.

Литерал ассоциативного массива задается так:

```
auto famousNamedConstants =
[ "пи" : 3.14, "e" : 2.71, "константа дивана" : 2.22 ];
```

Каждая ячейка литерала ассоциативного массива имеет вид ключ: значение. Тип ключей литерала ассоциативного массива вычисляется по массиву, в который неявно записываются все эти ключи, с помощью описанного выше способа. Тип значений вычисляется аналогично. После вычисления типа ключей K и типа значений V литерал типизируется как V[K]. Например, константа famousNamedConstants принимает тип double[string].

2.2.7. Функциональные литералы

В некоторых языках имя функции задается в ее определении; впоследствии такие функции вызываются по именам. Другие языки предоставляют возможность определить анонимную функцию (так называемую *лямбда-функцию*) прямо там, где она должна использоваться. Такое средство помогает строить мощные конструкции, задействующие функции более высокого порядка, то есть функции, принимающие в качестве аргументов и/или возвращающие другие функции. Функциональные литералы D позволяют определять анонимные функции *in situ*[2] – когда бы ни ожидалось имя функции.

Задача этого раздела – всего лишь показать на нескольких интересных примерах, как определяются функциональные литералы. Примеры более действенного применения этого мощного средства отложим до главы 5. Вот базовый синтаксис функционального литерала:

```
auto f = function double(int x) { return x / 10.; };
auto a = f(5);
assert(a == 0.5);
```

---

[1] Заключенное в 1989 году соглашение между коммунистами и демократами, ознаменовавшее собой достижение компромисса между двумя партиями. В данном случае также ищется «компромиссный» тип. – *Прим. пер.*

[2] In situ (лат.) – на месте. – *Прим. пер.*

Функциональный литерал определяется по тем же синтаксическим правилам, что и функция, с той лишь разницей, что определению предшествует ключевое слово function, а имя функции отсутствует. Рассмотренный пример в общем-то даже не использует анонимность, так как анонимная функция немедленно связывается с идентификатором f. Тип f – «указатель на функцию, принимающую int и возвращающую double». Этот тип записывается как double function(int) (обратите внимание: ключевое слово function и возвращаемый тип поменялись местами), так что эквивалентное определение f выглядит так:

```
double function(int) f = function double(int x) { return x / 10.; };
```

Кажущаяся странной перестановка function и double на самом деле сильно облегчает всем жизнь, позволяя отличить функциональный литерал по типу. Формулировка для легкого запоминания: слово function стоит в начале определения литерала, а в типе функции замещает имя функции.

Для простоты в определении функционального литерала можно опустить возвращаемый тип – компилятор определит его для вас по контексту, ведь ему тут же доступно тело функции.

```
auto f = function(int x) { return x / 10.; };
```

Наш функциональный литерал использует только собственный параметр x, так что его значение можно выяснить, взглянув лишь на тело функции и не принимая во внимание окружение, в котором она используется. Но что если функциональному литералу потребуется использовать данные, которые присутствуют в точке вызова, но не передаются как аргумент? В этом случае нужно заменить слово function словом delegate:

```
int c = 2;
auto f = delegate double(int x) { return c * x / 10.; };
auto a = f(5);
assert(a == 1);
c = 3;
auto b = f(5);
assert(b == 1.5);
```

Теперь тип f – delegate double(int x). Все правила распознавания типа для function применимы без изменений к delegate. Отсюда справедливый вопрос: если конструкции delegate могут делать все, на что способны function-конструкции (в конце концов конструкции delegate *могут*, но *не обязаны* использовать переменные своего окружения), зачем же сначала возиться с функциями? Нельзя ли всегда использовать конструкции delegate? Ответ прост: все дело в эффективности. Очевидно, что конструкции delegate обладают доступом к большему количеству информации, а по непреложному закону природы за такой доступ приходится расплачиваться. На самом деле, размер function равен размеру

указателя, а delegate – в два раза больше (один указатель на функцию, один – на окружение).

2.3. Операции

В следующих главах подробно описаны все операторы D в порядке убывания приоритета. Это естественный порядок, в котором вы бы группировали и вычисляли небольшие подвыражения в группах все большего размера.

С операторами тесно связаны две независимые темы: l- и r-значения и правила преобразования чисел. Необходимые определения приведены в следующих двух разделах.

2.3.1. L-значения и r-значения

Множество операторов срабатывает только тогда, когда l-значения удовлетворяют ряду условий. Например, не нужно быть гением, чтобы понять: присваивание 5 = 10 не соответствует правилам. Для успеха присваивания необходимо, чтобы левый операнд был *l-значением*. Пора дать точное определение l-значения (а заодно и сопутствующего ему *r-значения*). Названия терминов происходят от реального положения этих значений относительно оператора присваивания. Например, в инструкции a = b значение a расположено слева от оператора присваивания, поэтому оно называется **l-значением**; соответственно значение b, расположенное справа, – это **r-значение**[1].

К l-значениям относятся:

- все переменные, включая параметры функций, даже те, которые запрещено изменять (то есть определенные с квалификатором immutable);
- элементы массивов и ассоциативных массивов;
- поля структур и классов (о них мы поговорим позже);
- возвращаемые функциями значения, помеченные ключевым словом ref (о них мы поговорим еще позже);
- разыменованные указатели.

Любое l-значение может выступить в роли r-значения. К r-значениям также относится все, что не вошло в этот список: литералы, перечисляемые значения (которые вводятся с помощью ключевого слова enum; см. раздел 7.3) и результаты таких выражений, как x + 5. Обратите внимание: для присваивания быть l-значением необходимо, но не достаточно – нужно успешно пройти еще несколько семантических проверок, таких как проверка прав на доступ (см. главу 6) и проверка прав на изменение (см. главу 8).

---

[1] От англ. left-value и right-value. – *Прим. науч. ред.*

2.3.2. Неявные преобразования чисел

Мы только что коснулись темы преобразований; теперь пора рассмотреть ее подробнее. Здесь достаточно запомнить всего несколько простых правил:

1. Если числовое выражение компилируется в C и *также* компилируется в D, то его тип будет одинаковым в обоих языках (обратите внимание: D не обязан принимать все выражения на C).
2. Никакое целое значение не преобразуется к типу меньшего размера.
3. Никакое значение с плавающей запятой не преобразуется неявно в целое значение.
4. Любое числовое значение (целое или с плавающей запятой) неявно преобразуется к любому значению с плавающей запятой.

Правило 1 лишь незначительно усложняет работу компилятора, и это обоснованное усложнение. Поскольку D достаточно сильно «пересекается» с C и C++, это вдохновляет людей на бездумное копирование целых функций на этих языках в программы на D. Так пусть уж лучше D из соображений безопасности и переносимости отказывается время от времени от некоторых конструкций, чем если бы компилятор «проглотил» модуль из 2000 строк, а полученная программа заработала бы не так, как ожидалось, что определенно осложнило бы жизнь незадачливому программисту. Однако с помощью правила 2 язык D закручивает гайки посильнее, чем C и C++. Так что при переносе кода из этих языков на D диагностирующие сообщения время от времени будут указывать вам на «сырые» куски кода, рекомендуя вставить подходящие проверки и явные преобразования типов.

Рисунок 2.3 иллюстрирует правила преобразования для всех числовых типов. Для преобразования выбирается кратчайший путь; для двух путей одинаковой длины результаты преобразований совпадают. Независимо от количества шагов преобразование считается одношаговым процессом, преобразования неупорядочены и им не назначены приоритеты – тип или преобразуется к другому типу, или нет.

2.3.2.1. Распространение интервала значений

В соответствии с приведенными выше правилами обыкновенное число, такое как 42, будет недвусмысленно оценено как число типа `int`. А теперь взгляните на столь же заурядную инициализацию:

```
ubyte x = 42;
```

По неумолимым законам проверки типов вначале 42 распознается как `int`. Затем это число типа `int` будет присвоено переменной `x`, а это уже влечет насильственное преобразование типов. Разрешать такое грубое преобразование опасно (ведь многие значения типа `int` на самом деле не поместятся в `ubyte`). С другой стороны, требовать преобразования типов для очевидно безошибочного кода было бы очень неприятно.

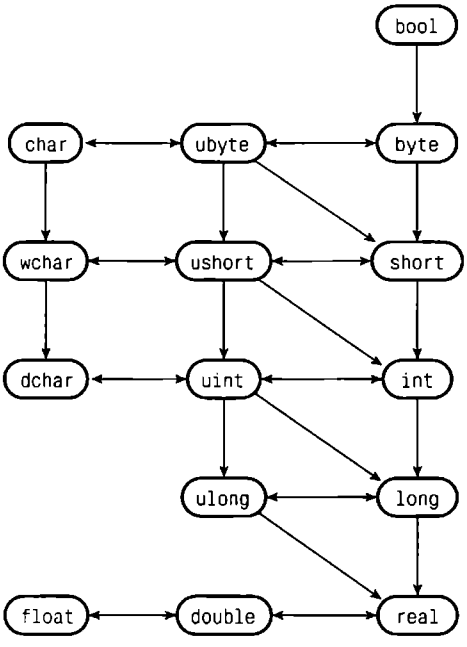

***Рис. 2.3.** Неявные преобразования чисел. Значение одного типа может быть автоматически преобразовано в значение другого типа тогда и только тогда, когда существует направленный путь от исходного типа до желаемого. Выбирается кратчайший путь, и преобразование считается одношаговым независимо от действительной длины пути. Преобразование в обратном направлении возможно, если оно осуществимо на основе метода распространения интервала значений (см. раздел 2.3.2.1)*

Язык D элегантно разрешает эту проблему с помощью способа, прообразом которого послужила техника оптимизации компиляторов, известная как *распространение интервала значений* (*value range propagation*): каждому значению в выражении ставится в соответствие интервал с границами в виде наименьшего и наибольшего возможных значений. Эти границы отслеживаются во время компиляции. Компилятор разрешает присвоить значение некоторого типа значению более «узкого» типа тогда и только тогда, когда интервальная оценка присваиваемого значения покрывается «целевым» типом. Очевидно, что для такой константы, как 42, как наибольшим, так и наименьшим значением будет 42, поэтому для присваивания нет преград.

Конечно же, в такой типовой ситуации можно было бы использовать гораздо более простой алгоритм, однако в общем случае логично применять метод распространения интервала значений, так как он прекрасно справляется и со сложными ситуациями. Рассмотрим функцию, которая извлекает из значения типа `int` младший и старший байты:

```
void fun(int val) {
   ubyte lsByte = val & 0xFF;
   ubyte hsByte = val >>> 24;

}
```

Этот код корректен независимо от того, каким будет введенное значение val. В первом выражении на значение накладывается маска, сбрасывающая все биты его старшего байта, а во втором делается сдвиг, в результате которого старший байт val перемещается на место младшего, а оставшиеся биты обнуляются.

И в самом деле, компилятор правильно типизирует функцию fun, так как сначала он вычисляет интервал val & 0xFF и получает [0; 255] независимо от val, затем вычисляет интервал для val >>> 24 и получает то же самое. Если бы вместо этих операций вы поставили операции, результат которых необязательно вместится в ubyte (например, val & 0x1FF или val >>> 23), компилятор не принял бы такой код.

Метод распространения интервала значений применим для всех арифметических и логических операций; например, значение типа uint, разделенное на 100000, всегда вместится в ushort. Кроме того, этот метод правильно работает и со сложными выражениями, такими как маскирование, после которого следует деление. Например:

```
void fun(int val) {
   ubyte x = (val & 0xF0F0) / 300;

}
```

В приведенном примере оператор & устанавливает границы интервала в 0 и 0xF0F0 (то есть 61680 в десятичной системе счисления). Затем операция деления устанавливает границы в 0 и 205. Любое число из этого диапазона вмещается в ubyte.

Определение корректности преобразований к меньшему типу по методу распространения интервала значений – несовершенный и консервативный подход. Одна из причин в том, что интервалы значений отслеживаются близоруко, внутри одного выражения, а не в нескольких смежных выражениях. Например:

```
void fun(int x) {
   if (x >= 0 && x < 42) {
      ubyte y = x; // Ошибка!
                   // Нельзя втиснуть int в ubyte!
      ...
   }
}
```

Совершенно ясно, что инициализация не содержит ошибок, но компилятор не поймет этого. Он бы мог, но это серьезно усложнило бы реализацию и замедлило процесс компиляции. Выбор был сделан в пользу

менее чувствительного распространения интервала значений в рамках одного выражения. Проведенный нами опыт показал, что такой умеренный анализ помогает программе избежать самых грубых ошибок, возникающих из-за ненадлежащего преобразования типов. Для оставшихся ошибок первого рода вы можете использовать выражения cast (см. раздел 2.3.6.7).

2.3.3. Типы числовых операций

В следующих разделах представлены операторы, применимые к числовым типам. Тип значения как результат различных операций с числами определяется с помощью нескольких правил. Это не лучшие правила, которые можно было бы придумать, но они достаточно просты, единообразны и систематичны.

В результате унарной операции всегда получается тот же тип, что и у операнда, кроме случая с оператором отрицания ! (см. раздел 2.3.6.6), применение которого всегда дает значения типа bool. Тип результата бинарных операций рассчитывается так:

- Если хотя бы один из операндов – число с плавающей запятой, то тип результата – наибольший из задействованных типов с плавающей запятой.
- Иначе если хотя бы один из операндов имеет тип ulong, то другой операнд до выполнения операции неявно преобразуется к типу ulong и результат также имеет тип ulong.
- Иначе если хотя бы один из операндов имеет тип long, то другой операнд до выполнения операции неявно преобразуется к типу long и результат также имеет тип long.
- Иначе если хотя бы один из операндов имеет тип uint, то другой операнд до выполнения операции неявно преобразуется к типу uint и результат также имеет тип uint.
- Иначе оба операнда до выполнения операции неявно преобразуются к типу int и результат имеет тип int.

Для всех неявных преобразований выбирается кратчайший путь (см. рис. 2.3). Это важная деталь. Например:

```
ushort x = 60_000;
assert(x / 10 == 6000);
```

В операции деления 10 имеет тип int и в соответствии с указанными правилами x неявно преобразуется к типу int до выполнения операции. На рис. 2.3 есть несколько возможных путей, в том числе прямое преобразование ushort → int и более длинное (на один шаг) ushort → short → int. Второе нежелательно, так как преобразование числа 60000 к типу short породит значение -5536, которое затем будет расширено до int и приведет инструкцию assert к ошибке. Выбор кратчайшего пути в графе преобразований помогает лучше защитить значение от порчи.

2.3.4. Первичные выражения

Первичные выражения – элементарные частицы вычислений. Нам уже встречались идентификаторы (см. раздел 2.1), логические литералы `true` и `false` (см. раздел 2.2.1), целые литералы (см. раздел 2.2.2), литералы с плавающей запятой (см. раздел 2.2.3), знаковые литералы (см. раздел 2.2.4), строковые литералы (см. раздел 2.2.5), литералы массивов (см. раздел 2.2.6) и функциональные литералы (см. раздел 2.2.7); все это первичные выражения, так же как и литерал `null`. В следующих разделах описаны другие первичные подвыражения: `assert`, `mixin`, `is` и выражения в круглых скобках.

2.3.4.1. Выражение assert

Некоторые выражения и инструкции, включая `assert`, используют нотацию *ненулевых* значений. Эти значения могут: 1) иметь числовой или знаковый тип (в этом случае смысл термина «ненулевое значение» очевиден), 2) иметь логический тип («ненулевое значение» интерпретируется как `true`) или 3) быть массивом, ссылкой или указателем (и тогда «ненулевым значением» считается не-`null`).

Выражение `assert(`*выражение*`)` вычисляет *выражение*. Если результат ненулевой, ничего не происходит. В противном случае выражение `assert` порождает исключение типа `AssertError`. Форма вызова `assert(`*выражение*`, `*сообщение*`)` делает *сообщение* (которое должно быть приводимо к типу `string`) частью сообщения об ошибке, хранимого внутри объекта типа `AssertError` (*сообщение* не вычисляется, если *выражение* ненулевое). Во всех случаях собственный тип `assert` – `void`.

Для сборки наиболее эффективного варианта программы компилятор D предоставляет специальный флаг (`-release` в случае эталонной реализации `dmd`), позволяющий игнорировать все выражения `assert` в компилируемом модуле (то есть вообще не вычислять *выражение*). Учитывая этот факт, к `assert` следует относиться как к инструменту отладки, а не как к средству проверки условий, поскольку оно может дать законный сбой. По той же причине некорректно использовать внутри выражений `assert` выражения с побочными эффектами, если поведение программы зависит от этих побочных эффектов. Более подробную информацию об итоговых сборках вы найдете в главе 11.

Ситуации, наподобие `assert(false)`, `assert(0)` и других, когда функция `assert` вызывается с заранее известным статическим нулевым значением, обрабатываются особым образом. Такие проверки всегда в силе (независимо от значений флагов компилятора) и порождают машинный код с инструкцией `HLT`, которая аварийно останавливает выполнение процесса. Такое прерывание может дать операционной системе подсказку сгенерировать дамп памяти или запустить отладчик с пометкой на виновной строке.

Предвосхищая рассказ о логических выражениях с логическим ИЛИ (см. раздел 2.3.15), упомянем простейшую концептуальную идею – всегда вычислять выражение и гарантировать его результат с помощью конструкции (выражение) || assert(false).

В главе 10 подробно обсуждаются механизмы обеспечения корректности программы, в том числе выражения assert.

2.3.4.2. Выражение mixin

Если бы выражения были отвертками разных видов, выражение mixin было бы электрической отверткой со сменными насадками, регулятором скоростей, адаптером для операций на мозге, встроенной беспроводной камерой и функцией распознавания речи. Оно на самом деле *такое* мощное.

Короче говоря, выражение mixin позволяет вам превратить строку в исполняемый код. Синтаксис выражения выглядит как mixin(выражение), где выражение должно быть строкой, известной во время компиляции. Это ограничение исключает возможность динамически создавать программный код, например читать строку с терминала и интерпретировать ее. Нет, D – не интерпретируемый язык, и его компилятор не является частью средств стандартной библиотеки времени исполнения. Хорошие новости заключаются в том, что D на самом деле запускает полноценный интерпретатор *во время компиляции*, а значит, вы можете собирать строки настолько изощренными способами, насколько этого требуют условия вашей задачи.

Возможность манипулировать строками и преобразовывать их в код во время компиляции позволяет создавать так называемые предметно-ориентированные встроенные языки программирования, которые их фанаты любовно обозначают аббревиатурой DSEL[1]. Типичный DSEL, реализованный на D, принимал бы инструкции в качестве строковых литералов, обрабатывал их в процессе компиляции, создавал соответствующий код на D в виде строки и с помощью mixin преобразовывал ее в готовый к исполнению код на D. Хорошим примером полезных DSEL могут служить SQL-команды, регулярные выражения и спецификации грамматик (а-ля yacc). На самом деле, даже вызывая printf, вы каждый раз используете DSEL. Спецификатор формата, применяемый функцией printf, – это настоящий маленький язык, ориентированный на описание шаблонов для текстовых данных.

D позволяет вам создать какой угодно DSEL без дополнительных инструментов (таких как синтаксические анализаторы, сборщики, генераторы кода и т. д.); например, функция bitfields из стандартной библиотеки (модуль std.bitmanip) принимает определения битовых полей и генериру-

[1] Domain-specific embedded language (DSEL) – предметно-ориентированный встроенный язык. – *Прим. пер.*

ет оптимальный код на D для их чтения и записи, хотя сам язык не поддерживает битовые поля.

2.3.4.3. Выражения is

Выражения is отвечают на вопросы о типах («Существует ли тип Widget?» или «Наследует ли Widget от Gadget?») и являются важной частью мощного механизма интроспекции во время компиляции, реализованного в D. Все выражения is вычисляются во время компиляции и возвращают логическое значение. Как показано ниже, есть несколько видов выражения is.

1. Выражения is(Тип) и is(Тип Идентификатор) проверяют, существует ли указанный Тип. Тип может быть недопустим или, гораздо чаще, просто не существует. Примеры:

```
bool
   a = is(int[]),    // True, int[] - допустимый тип
   b = is(int[5]),   // True, int[5] - также допустимый тип
   c = is(int[-3]),  // False, размер массива задан неверно
   d = is(Blah);     // False (если тип с именем Blah не был определен)
```

   Во всех случаях Тип должен быть записан корректно с точки зрения синтаксиса, даже если запись в целом лишена смысла; например, выражение is([[]x[]]) породит ошибку во время компиляции, а не вернет значение false. Другими словами, вы можете наводить справки только о том, что синтаксически выглядит как тип.

   Если присутствует Идентификатор, он становится псевдонимом типа Тип в случае истинности выражения is. Пока что неизвестная команда static if позволяет различать случаи истинности и ложности этого выражения. Подробное описание static if вы найдете в главе 3, но на самом деле все просто: static if вычисляет свое условие во время компиляции и позволяет компилировать вложенные в него инструкции, только если тестируемое выражение истинно.

```
static if (is(Widget[100][100] ManyWidgets)) {
   ManyWidgets lotsOfWidgets;
   ..
}
```

2. Выражения is(Тип1 == Тип2) и is(Тип1 Идентификатор == Тип2) возвращают True, если Тип1 и Тип2 идентичны. (Они могут иметь различные имена в результате применения alias.)

```
alias uint UInt;
assert(is(uint == UInt));
```

   Если присутствует Идентификатор, он становится псевдонимом типа Тип1 в случае истинности выражения is.

3. **Выражения** `is(Тип1 : Тип2)` **и** `is(Тип1 Идентификатор : Тип2)` **возвращают** `True`, **если** `Тип1` **идентичен или может быть неявно преобразован к типу** `Тип2`. **Например:**

```
bool
   a = is(int[5] : int[]),  // true, int[5] может быть
                            // преобразован к int[]
   b = is(int[5] == int[]), // FALSE; это разные типы
   c = is(uint : long),     // true
   d = is(ulong : long);    // true
```

   Аналогично, если присутствует `Идентификатор`, **он становится псевдонимом типа** `Тип1` **в случае истинности выражения** `is`.

4. **Выражения** `is(Тип == Вид)` **и** `is(Тип Идентификатор == Вид)` **проверяют, принадлежит ли** `Тип` **к категории** `Вид`. `Вид` **— это одно из следующих ключевых слов:** `struct`, `union`, `class`, `interface`, `enum`, `function`, `delegate`, `super`, `const`, `immutable`, `inout`, `shared` **и** `return`. **Выражение** `is` **истинно, если** `Тип` **соответствует указанному** `Виду`. **Если присутствует** `Идентификатор`, **он должен быть задан в зависимости от значения** `Вид` **(табл. 2.4).**

Таблица 2.4. Зависимости для значения `Идентификатор` *в выражении* `is(Тип Идентификатор == Вид)`

Вид	Идентификатор — псевдоним для...
`struct`	`Тип`
`union`	`Тип`
`class`	`Тип`
`interface`	`Тип`
`enum`	**Базовый тип перечисления (см. главу 7)**
`function`	**Кортеж типов аргументов функции**
`delegate`	**Функциональный тип** `delegate`
`super`	**Родительский класс (см. главу 6)**
`const`	`Тип`
`immutable`	`Тип`
`inout`	`Тип`
`shared`	`Тип`
`return`	**Тип, возвращаемый функцией, оператором** `delegate` **или указателем на функцию**

2.3.4.4. Выражения в круглых скобках

Круглые скобки переопределяют обычный порядок выполнения операций: для любых выражений, (<выражение>) обладает более высоким приоритетом, чем <выражение>.

2.3.5. Постфиксные операции

2.3.5.1. Доступ ко внутренним элементам

Оператор доступа ко внутренним элементам a.b предоставляет доступ к элементу с именем b, расположенному внутри объекта или типа a. Если a – сложное значение или сложный тип, допустимо заключить его в круглые скобки. В качестве b также может выступать выражение с ключевым словом new (см. главу 6).

2.3.5.2. Увеличение и уменьшение на единицу

Постфиксный вариант операции увеличения и уменьшения на единицу (значение++ и значение-- соответственно) определен для всех числовых типов и указателей и имеет тот же смысл, что и одноименная операция в C и C++: применение этой операции увеличивает или уменьшает на единицу значение (которое должно быть l-значением), возвращая копию этого значения до его изменения. (Аналогичный префиксный вариант операции увеличения и уменьшения на единицу описан в разделе 2.3.6.3.)

2.3.5.3. Вызов функции

Уже знакомый оператор вызова функции fun() инициирует выполнение кода функции fun. Синтаксис fun(<*список аргументов, разделенных запятыми*>) передает в тело fun список аргументов. Все аргументы вычисляются слева направо перед вызовом fun. Количество и типы значений в списке аргументов должны соответствовать количеству и типам формальных параметров. Если функция определена с атрибутом @property, то указание просто имени функции эквивалентно вызову этой функции без аргументов. Обычно fun – это имя функции, указанное в ее определении, но может быть и функциональным литералом (см. раздел 2.2.7) или выражением, возвращающим указатель на функцию или delegate. Подробно функции описаны в главе 5.

2.3.5.4. Индексация

Выражение arr[i] позволяет получить доступ к i-му элементу массива или ассоциативного массива arr (элементы массива индексируются начиная с 0). Если массив неассоциативный, то значение i должно быть целым. Иначе значение i должно иметь тип, который может быть неявно преобразован к типу ключа массива arr. Если индексирующее выражение находится слева от оператора присваивания (например, arr[i] = e)

и arr – ассоциативный массив, выполняется вставка элемента в массив, если его там не было. Иначе если i относится к элементу, которого нет в массиве arr, выражение порождает исключение типа RangeError. В качестве arr и i также могут выступать указатель и целое соответственно. Операции индексации с помощью указателей автоматически не проверяются. В некоторых режимах сборки (небезопасные итоговые сборки; см. раздел 4.1.2) отменяется проверка границ и в случае неассоциативных массивов.

2.3.5.5. Срезы массивов

Если arr – линейный (неассоциативный) массив, выражение arr[i .. j] возвращает массив, ссылающийся на интервал внутри arr от i-го до j-го элемента (не включая последний). Значения i и j, отмечающие границы среза, должны допускать неявное преобразование в целое. Выражение arr[] позволяет адресовать срез массива величиной в целый массив arr. Данные не копируются «по-настоящему», поэтому изменение среза массива влечет к изменению содержимого исходного массива arr. Например:

```
int[] a = new int[5]; // Создать массив из пяти целых чисел
int[] b = a[3    5];  // b ссылается на два последних элемента a
b[0] = 1;
b[1] = 3;
assert(a == [ 0, 0, 0, 1, 3 ]); // a был изменен
```

Если i > j или j > a.length, генерируется исключение типа RangeError. Иначе если i == j, будет возвращен пустой массив. В качестве arr в выражении arr[i .. j] можно использовать указатель. В этом случае будет возвращен массив, отражающий область памяти начиная с адреса arr + i до arr + j (не включая элемент с адресом arr + j). Если i > j, генерируется ошибка RangeError, иначе при получении среза указателя границы не проверяются. И снова в некоторых режимах сборки (небезопасные итоговые сборки, см. раздел 4.1.2) все проверки границ при получении срезов могут быть отключены.

2.3.5.6. Создание вложенного класса

Выражение вида a.new T, где a – значение типа class, создает объект типа T, чье определение вложено в определение a. Что-то непонятно? Это потому что мы еще не определили ни классы, ни вложенные классы, и даже сами выражения new пока не рассмотрели. Определение выражения new уже совсем близко (в разделе 2.3.6.1), а чтобы познакомиться с определениями классов и вложенных классов, придется подождать до главы 6 (точнее до раздела 6.11). А до тех пор считайте этот раздел просто заглушкой, необходимой для целостности изложения.

2.3.6. Унарные операции

2.3.6.1. Выражение new

Допустимы несколько вариантов выражения new:

```
new (‹адрес›)опц ‹Тип›
new (‹адрес›)опц ‹Тип›(‹список_аргументов›опц)
new (‹адрес›)опц ‹Тип›[‹список_аргументов›]
new (‹адрес›)опц ‹Анонимный класс›
```

Забудем на время про необязательный (<*адрес*>). Два первых варианта new T и new T(<*список_аргументов*>$_{опц}$) динамически выделяют память для объекта типа T. Второй вариант позволяет передать аргументы конструктору T. (Формы new T и new T() тождественны друг другу и создают объект, инициализированный по умолчанию.) Мы пока что не рассматривали типы с конструкторами, поэтому давайте отложим этот разговор до главы 6, предоставляющей подробную информацию о классах (см. раздел 6.3), и главы 7, посвященной пользовательским типам (см. раздел 7.1.3). Также отложим вопрос создания анонимных классов (последний в списке вариант выражения new), см. раздел 6.11.3.

А здесь сосредоточимся на создании уже хорошо известных массивов. Выражение new T[n] выделяет непрерывную область памяти, достаточную для размещения n объектов типа T подряд, заполняет эти места значениями T.init и возвращает ссылку на них в виде значения типа T[]. Например:

```
auto arr = new int[4];
assert(arr.length == 4);
assert(arr == [ 0, 0, 0, 0 ]); // Инициализирован по умолчанию
```

Тот же результат можно получить, чуть изменив синтаксис:

```
auto arr = new int[](4);
```

На этот раз выражение интерпретируется как new T(4), где под T понимается int[]. Опять же результатом будет массив из четырех элементов, доступ к которому предоставляет переменная arr типа int[].

У второго варианта действительно больше возможностей, чем у первого. Если требуется выделить память под массив массивов, в круглых скобках можно указать несколько аргументов. Эти значения инициализируют массивы по строкам. Например, память под массив, состоящий из четырех массивов по восемь элементов, можно выделить так:

```
auto matrix = new int[][](4, 8);
assert(matrix.length == 4);
assert(matrix[0].length == 8);
```

Первая строка в рассмотренном коде заменяет более многословную запись:

```
auto matrix = new int[][](4);
foreach (ref row; matrix) {
   row = new int[](8);
}
```

Во всех рассмотренных случаях память выделяется из кучи с автоматической сборкой мусора. Память, которая больше не используется и недоступна программе, отправляется обратно в кучу. Библиотека времени исполнения из эталонной реализации предоставляет множество специализированных средств для управления памятью в модуле core.gc, в том числе изменение размера только что выделенного блока памяти и освобождение памяти вручную. Управление памятью вручную – рискованное занятие, поэтому всегда избегайте его, кроме случаев, когда это абсолютно необходимо.

Необязательный адрес, расположенный сразу после ключевого слова new, вводит конструкцию, называемую *новым размещением*. По смыслу вариант new(адрес) T отличается от других: вместо выделения памяти под новый объект происходит размещение объекта по заданному адресу. Такие низкоуровневые средства в обычном коде не применяются. Вы можете использовать их, например, чтобы распределять память из кучи C с помощью malloc и затем использовать ее для хранения значений языка D.

2.3.6.2. Получение адреса и разыменование

Поскольку мы еще будем говорить об указателях, сейчас упомянем лишь о парных операторах получения адреса и разыменования. Выражение &значение получает адрес значения (которое должно быть l-значением) и возвращает указатель с типом T*, если значение имеет тип T.

Обратная операция *p разыменовывает указатель, отменяя операцию получения адреса; выражение *&значение преобразуется к виду значение. Подробный разговор об указателях намеренно отложен до главы 7, потому что в D можно многого добиться и без использования указателей – низкоуровневых и опасных средств языка.

2.3.6.3. Увеличение и уменьшение на единицу (префиксный вариант)

Выражения ++значение и --значение соответственно увеличивают и уменьшают значение (которое должно быть числом или указателем) на единицу, возвращая в качестве результата только что измененное значение.

2.3.6.4. Поразрядное отрицание

Выражение ~a инвертирует (изменяет на противоположное значение) каждый бит в a и возвращает значение того же типа, что и a. a должно быть целым числом.

2.3.6.5. Унарный плюс и унарный минус

Выражение +значение не делает ничего особенного: оператор унарный плюс включен в язык лишь из соображений сохранения целостности. Выражение -значение равносильно выражению 0 - значение; унарный минус используется только с операндом-числом.

Одна из странностей поведения унарного минуса: применив этот оператор к числу без знака, получим также число без знака (по правилам, изложенным в разделе 2.3.3), например -55u – это 4_294_967_241, то есть uint.max - 55 + 1.

То, что числа без знака на самом деле не являются натуральными, – суровая правда жизни. Для D, как и для других языков, двоичная арифметика со своими простыми правилами переполнения – неизбежная реальность, от которой пользователя не защитят никакие абстракции. Один из способов не ошибиться в трактовке выражения -значение, где значение – любое целое число, – считать его краткой записью ~значение + 1; другими словами, инвертировать каждый из разрядов значения и прибавить 1 к полученному результату. Такая процедура не вызывает вопросов, связанных с наличием знака у типа переменной значение.

2.3.6.6. Отрицание

Выражение !значение имеет тип bool и возвращает false, если значение ненулевое (определение ненулевого значения приведено в разделе 2.3.4.1), иначе возвращается true.

2.3.6.7. Приведение типов

Оператор приведения типов подобен могущественному доброму Джинну из лампы, который всегда рад выручить. При этом, как и герой мультфильма, он своенравен, чуть глуховат и не упустит случая развлечься, слишком буквально выполняя нечетко сформулированные желания, что обычно приводит к катастрофе.

Несмотря на сказанное, в редких случаях приведение типов бывает полезно, если система статической типизации недостаточно проницательна, чтобы отследить все ваши «эксплойты». Выглядит приведение типов так: cast(Тип) a.

Перечислим виды приведения типов по убыванию безопасности использования.

- *Приведение ссылок* – преобразование между ссылками на объекты class и interface. Такие приведения всегда динамически проверяются.
- *Приведение чисел* – принудительное преобразование данных любого числового типа в данные любого другого числового типа.

- *Приведение массивов* – преобразование между разными типами массивов; общий размер исходного массива должен быть кратен размеру элементов целевого массива.
- *Приведение указателей* – преобразование указателя одного типа в указатель другого типа.
- *Приведение указатель/число* – перевод указателя в целый тип достаточного размера, чтобы вместить этот указатель, и наоборот.

Будьте предельно осторожны со всеми непроверяемыми приведениями типов, особенно с тремя последними, поскольку они могут нарушить целостность системы типов.

2.3.7. Возведение в степень

Синтаксис выражения возведения в степень: основание `^^` показатель (основание возводится в степень показатель). И основание, и показатель должны быть числами. То же самое делает функция pow(основание, показатель), которую можно найти в стандартных библиотеках языков C и D (обратитесь к документации для своего модуля `std.math`). Тем не менее запись некоторых числовых выражений действительно выигрывает от синтаксического упрощения.

Результат возведения нуля в нулевую степень – единица, а в любую другую – ноль.

2.3.8. Мультипликативные операции

К мультипликативным операциям относятся умножение (`a * b`), деление (`a / b`) и получение остатка от деления (`a % b`). Они применимы исключительно к числовым типам.

Если в целочисленных операциях `a / b` или `a % b` в качестве `b` участвует ноль, будет сгенерирована аппаратная ошибка. Дробный результат деления всегда округляется в меньшую сторону (например, в результате `7 / 3` получим 2, а результатом `-7 / 3` будет `-1`). Операция `a % b` определена так, что `a == (a / b) * b + a % b`, поэтому в результате `7 % 3` получим `1`, а результатом `-7 % 3` будет `-1`.

В языке D также можно определить остаток от деления для чисел с плавающей запятой. Это более запутанное определение. Если в выражении `a % b` в качестве `a` или `b` выступает число с плавающей запятой, результатом становится наибольшее (по модулю) число с плавающей запятой `r`, удовлетворяющее следующим условиям:

- `a` и `r` не имеют противоположных знаков;
- `r` меньше `b` по модулю, то есть `abs(r) < abs(b)`;
- существует такое целое число `q`, что `r == a - q * b`.

Если такое число найти невозможно, результатом `a % b` будет особое значение NaN.

2.3.9. Аддитивные операции

Аддитивные операции – это сложение a + b, вычитание a - b и конкатенация a ~ b.

Сложение и вычитание применимы только к числам. Тип результата определяется в соответствии с правилами из раздела 2.3.3.

Операция конкатенации может быть применена к операндам a и b, если хотя бы один из них является массивом элементов некоторого типа T. В качестве другого операнда должен выступать либо массив элементов типа T, либо значение типа, неявно преобразуемого к типу T. В результате получается новый массив, созданный из размещенных друг за другом a и b.

2.3.10. Сдвиг

В языке D есть три операции сдвига, в каждой из которых участвуют два целочисленных операнда: a << b, a >> b и a >>> b. Во всех случаях значение b должно иметь тип без знака; значение со знаком необходимо привести к значению беззнакового типа (разумеется, предварительно убедившись, что b >= 0; результат сдвига на отрицательное количество разрядов непредсказуем). a << b сдвигает a влево (то есть в направлении самого старшего разряда a) на b бит, а a >> b сдвигает a вправо на b бит. Если a – отрицательное число, знак после сдвига сохраняется.

a >>> b – это беззнаковый сдвиг независимо от знаковости a. Это означает, что ноль гарантированно займет самый старший разряд a. Проиллюстрируем сюрпризы, которые готовит применение операции сдвига к числам со знаком:

```
int a = -1;                 // То есть 0xFFFF_FFFF
int b = a << 1;
assert(b == -2);            // 0xFFFF_FFFE
int c = a >> 1;
assert(c == -1);            // 0xFFFF_FFFF
int d = a >>> 1;
assert(d == +2147483647); // 0x7FFF_FFFF
```

Сдвиг на число разрядов большее, чем в типе a, запрещается во время компиляции, если b – статически заданное, заранее известное значение. Если же b определяется во время исполнения программы, то результат такого сдвига зависит от реализации компилятора:

```
int a = 50;
uint b = 35;
a << 33;          // Ошибка во время компиляции
auto c = a << b; // Результат зависит от реализации
auto d = a >> b; // Результат зависит от реализации
```

В любом случае тип результата определяется в соответствии с правилами из раздела 2.3.3.

Раньше было популярно с помощью операции сдвига реализовывать быстрое целочисленное умножение на 2 (a << 1) или деление на 2 (a >> 1) – или в общем случае умножение и деление на различные степени 2. Эта техника вышла из употребления, подобно видеокассетам. Пишите просто: a * k или a / k; если значение k известно на этапе компиляции, компилятор гарантированно сгенерирует для вас оптимальный код с операциями сдвига и всем, что еще нужно, избавив вас от волнений по поводу тонкостей работы со знаком. Не ищите сдвига на свою голову.

2.3.11. Выражения in

Если ключ – это значение типа K, а массив – ассоциативный массив типа V[K], то выражение вида ключ in массив порождает значение типа V* (указатель на V). Если ассоциативный массив содержит пару (ключ, значение), то указатель указывает на значение. В противном случае полученный указатель – null.

Для обратной, отрицательной проверки вы, конечно, можете написать !(ключ in массив), но есть и более сжатая форма – ключ !in массив – с тем же приоритетом, что и у ключ in массив.

Зачем нужны все эти сложности с указателями, когда выражение a in b может просто возвращать значение логического типа? Ответ: для эффективности. Довольно часто требуется узнать, есть ли в массиве нужный индекс, и если есть, то использовать соответствующий ему элемент. Можно написать что-то вроде:

```
double[string] table;
  .
if ("hello" in table) {
   ++table["hello"];
} else {
   table["hello"] = 0;
}
```

Проблема этого кода в том, что он дважды обращается к массиву в случае, если запрошенный индекс найден. Используя возвращенный указатель, можно создавать более эффективный код, например:

```
double[string] table;
..
auto p = "hello" in table;
if (p) {
   ++*p;
} else {
   table["hello"] = 1;
}
```

2.3.12. Сравнение

2.3.12.1. Проверка на равенство

Операция вида `a == b`, возвращающая значение типа `bool`, имеет следующую семантику. Во-первых, если операнды имеют разный тип, сначала они неявно преобразуются к одному типу. Затем операнды проверяются на равенство следующим образом:

- для целых чисел и указателей выполняется точное поразрядное сравнение операндов;
- для чисел с плавающей запятой приняты правила: `-0` считается равным `+0`, а `NaN` – не равным `NaN`[1]; во всех остальных случаях операнды сравниваются поразрядно;
- равенство объектов типа `class` определяется с помощью оператора `opEquals` (см. раздел 6.8.3);
- равенство массивов означает поэлементное равенство;
- по умолчанию равенство объектов типа `struct` определяется как равенство всех полей операндов; пользовательские типы могут переопределять это поведение (см. главу 12).

Операция вида `a != b` служит для проверки на неравенство.

С помощью выражения `a is b` проверяется *равенство ссылок* (*alias equality*): если `a` и `b` ссылаются на один и тот же объект, выражение возвращает `true`:

- если `a` и `b` – массивы или ссылки на классы, результатом будет `true`, только если `a` и `b` – это два имени для одного реального объекта;
- в остальных случаях для `a` и `b` должно быть истинно выражение `a == b`.

Мы пока не касались классов, но пример с массивами может быть полезным:

```
import std.stdio;

void main() {
   auto a = "какая-то строка";
   auto b = a; // a и b ссылаются на один и тот же массив
   a is b && writeln("Ага, это действительно одно и то же.");
   auto c = "какая-то (другая) строка";
   a is c || writeln("Действительно.   не одно и то же.");
}
```

[1] Стандарт IEEE 754 определяет для чисел с плавающей запятой два разных двоичных представления для нуля: `-0` и `+0`. Это порождает ряд неудобств, таких как исключение при сравнении чисел, рассмотренное здесь. С другой стороны, скорость многих вычислений увеличивается. Вы, скорее всего, будете редко использовать литерал `-0.0` в коде на D, но это значение может получиться неявно как результат вычислений, асимптотически приближающих отрицательные значения к нулю.

Этот код печатает оба сообщения, потому что a и b связаны с одним и тем же массивом, в то время как c ссылается на другой массив. В общем случае возможно, чтобы два массива обладали одинаковым содержимым (тогда выражение a == b истинно), но они указывают на разные области памяти, поэтому проверка вида a is b вернет false. Разумеется, когда выражение a is b истинно, то и a == b также истинно (если только ваши планки оперативной памяти не куплены по дешевке).

Вместо выражения проверки на неравенство !(a is b) можно использовать его краткий вариант a !is b.

2.3.12.2. Сравнение для упорядочивания

В языке D определены стандартные логические операции вида a < b, a <= b, a > b и a >= b (меньше, меньше или равно, больше, больше или равно). При сравнении чисел одно из них должно быть неявно преобразовано к типу другого. Для операндов с плавающей запятой считается, что -0 равен 0, то есть результатом сравнения -0 < 0 будет false. Если хотя бы один из операндов равен NaN, то любая операция упорядочивающего сравнения вернет false (пусть это и кажется парадоксом).

Как обычно NaN портит добропорядочным числам с плавающей запятой весь праздник. Все сравнения, в которых участвует хотя бы одно значение NaN, порождают «исключение в операции с плавающей запятой» (floating-point exception). С точки зрения терминологии языков программирования это не совсем обычное исключение: возникает лишь особая ситуация на уровне аппаратного обеспечения, которую можно отдельно обработать. D предоставляет интерфейс для математического сопроцессора через модуль std.c.fenv.

2.3.12.3. Неассоциативность

Одно из важных свойств операторов сравнения в языке D – их *неассоциативность*. Любая цепочка операторов сравнения вида a <= b < c некорректна.

Простой способ определить операторы сравнения – сделать так, чтобы они возвращали значение типа bool. Возможность сравнивать логические значения друг с другом имеет не очень приятное следствие: смысл выражения a <= b < c не совпадает с привычным для маленького математика внутри нас, который то и дело пытается напомнить о себе. Вместо «b больше или равно a и меньше c» выражение будет распознано как (a <= b) < c, то есть «логический результат сравнения a <= b сравнить с c». Например, выражение 3 <= 4 < 2 было бы истинно! Такая семантика вряд ли желательна.

Можно было бы решить эту проблему, разрешив выражение a <= b < c и наделив его истинным математическим значением: a <= b && b < c, с тем чтобы b вычислялось только один раз. В языках Python и Perl 6

принята именно такая семантика, позволяющая использовать произвольные цепочки сравнений, такие как `a < b == c > d < e`. Но D – наследник не этих языков. Разрешение использовать выражения на C, но со слегка измененной семантикой (хотя Python и Perl 6, скорее всего, выбрали верное направление), добавило бы больше неразберихи, чем удобства, поэтому разработчики D решили просто-напросто запретить такую конструкцию.

2.3.13. Поразрядные ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и И

Выражения `a | b`, `a ^ b` и `a & b` представляют собой поразрядные операции ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и И соответственно. Перед выполнением операции вычисляются оба операнда (неполное вычисление логических выражений не допускается), даже если результат определяется уже по одному из них.

И `a`, и `b` должны быть целыми числами. Тип результата определяется в соответствии с правилами из раздела 2.3.3.

2.3.14. Логическое И

В свете вышесказанного неудивительно, что значение выражения `a && b` зависит от типа `b`.

- Если тип `b` не `void`, то результатом выражения будет логическое значение. Если операнд `a` ненулевой, вычисляется `b`, и только в том случае, если он также ненулевой, возвращается `true`, иначе возвращается `false`.
- Если `b` имеет тип `void`, то и все выражение имеет тип `void`. Если операнд `a` ненулевой, то вычисляется операнд `b`. Иначе `b` не вычисляется.

Оператор `&&` с выражением типа `void` в качестве правого операнда можно использовать в роли краткой инструкции `if`:

```
string line;

line == "#\n" && writeln("Успешно принята строка #. ");
```

2.3.15. Логическое ИЛИ

Семантика выражения `a || b` зависит от типа `b`.

- Если тип `b` не `void`, выражение имеет тип `bool`. Если операнд `a` ненулевой, выражение возвращает `true`. Иначе вычисляется `b`, и только в том случае, если он также ненулевой, возвращается `true`.
- Если `b` имеет тип `void`, то и все выражение имеет тип `void`. Если операнд `a` ненулевой, то операнд `b` не вычисляется. Иначе `b` вычисляется.

Второе правило можно применять для обработки непредсказуемых обстоятельств:

```
string line;

line.length > 0 || line = "\n";
```

2.3.16. Тернарная условная операция

Тернарная условная операция – это конструкция типа `if-then-else` с синтаксисом `a ? b : c`, с которой вы, возможно, знакомы. Если операнд `a` ненулевой, условное выражение вычисляется и возвращается `b`; иначе выражение вычисляется и возвращается `c`. Ценой героических усилий компилятор определяет наиболее «узкий» тип для `b` и `c`, который становится типом всего выражения. Этот тип (назовем его `T`) вычисляется с помощью простого алгоритма (показанного на примерах):

1. Если `b` и `c` одного типа, он выступает и в роли `T`.
2. Иначе если `b` и `c` – целые числа, сначала типы меньше 32 разрядов расширяются до `int`, затем `T` присваивается больший тип; при одинаковых размерах приоритет имеет тип без знака.
3. Иначе если один операнд – целого типа, а другой – с плавающей запятой, в качестве `T` выбирается тип с плавающей запятой.
4. Если оба операнда относятся к типу с плавающей запятой, то `T` – наибольший из этих типов.
5. Иначе если типы имеют один и тот же супертип (то есть базовый тип), `T` принимает вид этого супертипа (к этой теме мы вернемся в главе 6).
6. Иначе делается попытка неявно привести `c` к типу `b` и `b` к типу `c`; если удается только что-то одно, в роли `T` выступает тип, полученный в результате удачного приведения.
7. Иначе выражение содержит ошибку.

Более того, если `b` и `c` одного типа и являются l-значениями, результатом также будет l-значение, что позволяет написать:

```
int x = 5, y = 5;
bool which = true;
(which ? x : y) += 5;
assert(x == 10);
```

Многие концептуальные примеры обобщенного программирования используют тернарную операцию сравнения для нахождения общего типа двух значений.

2.3.17. Присваивание

Операции присваивания имеют вид `a = b` или `a ω= b`, где буква ω выступает в качестве одного из операторов `^^`, `*`, `/`, `%`, `+`, `-`, `~`, `<<`, `>>`, `>>>`, `|`, `^` или `&`, а также «за обязательное использование греческих букв в книгах по программированию». С применением самостоятельных вариантов этих операторов вы уже познакомились в предыдущих разделах.

Выражение a ω= b семантически идентично выражению вида a = a ω b, однако между этими формами записи все же есть серьезное различие: в первом случае a вычисляется всего один раз (представьте, что a и b – достаточно сложные выражения, например: array[i * 5 + j] *= sqrt(x)).

Независимо от приоритета ω оператор ω= имеет тот же приоритет, что и сам оператор =, то есть ниже, чем у оператора сравнения (о котором речь шла выше), и чуть выше, чем у запятой (о которой речь пойдет ниже). Также независимо от ассоциативности ω все операторы вида ω= (в том числе =) ассоциативны слева, например a /= b = c -= d – это то же самое, что и a /= (b = (c -= d)).

2.3.18. Выражения с запятой

Выражения, разделенные запятыми, вычисляются последовательно друг за другом. Результат выражения в целом – это результат самого правого подвыражения. Например:

```
int a = 5;
int b = 10;
int c = (a = b, b = 7, 8);
```

После выполнения этого фрагмента кода переменные a, b и c примут значения 10, 7 и 8 соответственно.

2.4. Итоги и справочник

На этом мы завершаем описание богатых возможностей языка D по построению выражений. В табл. 2.5. собраны все операторы языка D. Вы можете использовать ее в качестве краткого справочника.

Таблица 2.5. Выражения D порядке убывания приоритета

Выражение	Описание
<идентификатор>	Идентификатор (см. раздел 2.1)
.*<идентификатор>*	Идентификатор, доступный в пространстве имен модуля (в обход всех других пространств имен) (см. раздел 2.1)
this	Текущий объект внутри метода (см. раздел 2.1.1)
super	Направляет поиск идентификаторов и динамический поиск методов в пространство имен объекта-родителя (см. раздел 2.1.1)
$	Текущий размер массива (допустимо использовать $ внутри индексирующего выражения или выражения получения среза) (см. раздел 2.1.1)
null	«Нулевая» ссылка, массив или указатель (см. раздел 2.1.1)

Выражение	Описание
`typeid(T)`	Получить объект `TypeInfo`, ассоциированный с `T` (см. раздел 2.1.1)
`true`	Логическое значение «истина» (см. раздел 2.2.1)
`false`	Логическое значение «ложь» (см. раздел 2.2.1)
<число>	Числовой литерал (см. разделы 2.2.2 и 2.2.3)
<знак>	Знаковый литерал (см. раздел 2.2.4)
<строка>	Строковый литерал (см. раздел 2.2.5)
<массив>	Литерал массива (см. раздел 2.2.6)
<функция>	Функциональный литерал (см. раздел 2.2.7)
`assert(a)`	В режиме отладки, если `a` не является ненулевым значением, выполнение программы прерывается; в режиме итоговой сборки (release) ничего не происходит (см. раздел 2.3.4.1)
`assert(a, b)`	То же, но к сообщению об ошибке добавляется `b` (см. раздел 2.3.4.1)
`mixin(a)`	Выражение `mixin` (см. раздел 2.3.4.2)
<IsExpr>	Выражение `is` (см. раздел 2.3.4.3)
`( a )`	Выражение в круглых скобках (см. раздел 2.3.4.4)
`a.b`	Доступ к вложенным элементам (см. раздел 2.3.5.1)
`a++`	Постфиксный вариант операции увеличения на единицу (см. раздел 2.3.5.2)
`a--`	Постфиксный вариант операции уменьшения на единицу (см. раздел 2.3.5.2)
`a(`*<арг$_{опц}$>*`)`	Оператор вызова функции (*<арг$_{опц}$>* = необязательный список аргументов, разделенных запятыми) (см. раздел 2.3.5.3)
`a[`*<арг>*`]`	Оператор индексации (*<арг>* = список аргументов, разделенных запятыми) (см. раздел 2.3.5.4)
`a[]`	Срез в размере всего массива (см. раздел 2.3.5.5)
`a[b .. c]`	Срез (см. раздел 2.3.5.5)
`a.`*<выражение new>*	Создание экземпляра вложенного класса (см. раздел 2.3.5.6)
`&a`	Получение адреса (см. раздел 2.3.6.2)
`++a`	Префиксный вариант операции увеличения на единицу (см. раздел 2.3.6.3)
`--a`	Префиксный вариант операции уменьшения на единицу (см. раздел 2.3.6.3)

Таблица 2.5 (продолжение)

Выражение	Описание
`*a`	Разыменование (см. раздел 2.3.6.2)
`-a`	Унарный минус (см. раздел 2.3.6.5)
`+a`	Унарный плюс (см. раздел 2.3.6.5)
`!a`	Отрицание (см. раздел 2.3.6.6)
`~a`	Поразрядное отрицание (см. раздел 2.3.6.4)
`(T).a`	Доступ к статическим внутренним элементам
`cast(T) a`	Приведение выражения `a` к типу `T`
<выражение new>	Создание объекта (см. раздел 2.3.6.1)
`a ^^ b`	Возведение в степень (см. раздел 2.3.7)
`a * b`	Умножение (см. раздел 2.3.8)
`a / b`	Деление (см. раздел 2.3.8)
`a % b`	Получение остатка от деления (см. раздел 2.3.8)
`a + b`	Сложение (см. раздел 2.3.9)
`a - b`	Вычитание (см. раздел 2.3.9)
`a ~ b`	Конкатенация (см. раздел 2.3.9)
`a << b`	Сдвиг влево (см. раздел 2.3.10)
`a >> b`	Сдвиг вправо (см. раздел 2.3.10)
`a >>> b`	Беззнаковый сдвиг вправо (старший разряд сбрасывается независимо от типа и значения `a`) (см. раздел 2.3.10)
`a in b`	Проверка на принадлежность для ассоциативных массивов (см. раздел 2.3.11)
`a == b`	Проверка на равенство; все операторы этой группы неассоциативны; например, выражение `a == b == c` некорректно (см. раздел 2.3.12.1)
`a != b`	Проверка на неравенство (см. раздел 2.3.12.1)
`a is b`	Проверка на идентичность (`true`, если и только если `a` и `b` ссылаются на один и тот же объект) (см. раздел 2.3.12.1)
`a !is b`	То же, что `!(a is b)`
`a < b`	Меньше (см. раздел 2.3.12.2)
`a <= b`	Меньше или равно (см. раздел 2.3.12.2)
`a > b`	Больше (см. раздел 2.3.12.2)
`a >= b`	Больше или равно (см. раздел 2.3.12.2)
`a | b`	Поразрядное ИЛИ (см. раздел 2.3.13)

Выражение	Описание
a ^ b	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ (см. раздел 2.3.13)
a & b	Поразрядное И (см. раздел 2.3.13)
a && b	Логическое И (b может иметь тип void) (см. раздел 2.3.14)
a || b	Логическое ИЛИ (b может иметь тип void) (см. раздел 2.3.15)
a ? b : c	Тернарная условная операция; если операнд a имеет ненулевое значение, то b, иначе c (см. раздел 2.3.16)
a = b	Присваивание; все операторы присваивания этой группы ассоциативны справа; например a *= b += c – то же, что и a *= (b += c) (см. раздел 2.3.17)
a += b	Сложение «на месте»; выражения со всеми операторами вида a ω= b, работающими по принципу «вычислить и присвоить», вычисляются в следующем порядке: 1) a (должно быть l-значением), 2) b и 3) $a_1 = a_1 \omega b$, где a_1 – l-значение, получившееся в результате вычисления a
a -=b	Вычитание «на месте»
a *= b	Умножение «на месте»
a /= b	Деление «на месте»
a %=b	Получение остатка от деления «на месте»
a &=b	Поразрядное И «на месте»
a |= b	Поразрядное ИЛИ «на месте»
a ^= b	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ «на месте»
a ~= b	Конкатенация «на месте» (присоединение b к a)
a <<= b	Сдвиг влево «на месте»
a >>= b	Сдвиг вправо «на месте»
a >>>= b	Беззнаковый сдвиг вправо «на месте»
a, b	Последовательность выражений; выражения вычисляются слева направо, результатом операции становится самое правое выражение (см. раздел 2.3.18)

3

Инструкции

Эта глава содержит обязательные определения всех инструкций языка D. D наследует внешний вид и функциональность языков семейства C – в нем есть привычные инструкции `if`, `while`, `for` и другие. Наряду с этим D предлагает ряд новых интересных инструкций и некоторое усовершенствование старых. Если неизбежное перечисление с подробным описанием каждой инструкции заранее нагоняет на вас скуку, то вот вам несколько «отступлений» – любопытных отличий D от других языков.

Если вы желаете во время компиляции проверять какие-то условия, то вас, скорее всего, заинтересует инструкция `static if` (см. раздел 3.4). Ее возможности гораздо шире, чем просто настройка набора флагов; тем, кто каким-либо образом использует обобщенный код, `static if` принесет ощутимую пользу. Инструкция `switch` (см. раздел 3.5) выглядит и действует в основном так же, как и ее тезка из языка C, но оперирует также строками, позволяя одновременно сопоставлять целые диапазоны. Для корректной обработки небольших конечных множеств значений пригодится инструкция `final switch` (см. разд. 3.6), которая работает с перечисляемыми типами и заставит вас реализовать обработчик абсолютно для каждого возможного значения. Инструкция `foreach` (см. разделы 3.7.4 и 3.7.5) помогает организовывать последовательные итерации; классическая инструкция `for` предоставляет больше возможностей, но и более многословна. Инструкция `mixin` (см. раздел 3.12) вставляет заранее определенный шаблонный код. Инструкция `scope` (см. раздел 3.13) значительно облегчает написание корректного безотказного кода с правильной обработкой ошибок, заменяя сумбурную конструкцию `try/catch/finally`, которой иначе вам пришлось бы воспользоваться.

3.1. Инструкция-выражение

Как уже говорилось (см. раздел 1.2), достаточно в конце выражения поставить точку с запятой, чтобы получить инструкцию:

```
a = b + c;
transmogrify(a + b);
```

При этом не любое выражение можно превратить в инструкцию. Если инструкция, которая должна получиться, не выполняет никакого действия, например:

```
1 + 1 == 2; // Абсолютная истина
```

то компилятор диагностирует ошибку.

3.2. Составная инструкция

Составная инструкция – это (возможно, пустая) последовательность инструкций, заключенных в фигурные скобки. Инструкции исполняются по порядку. Скобки ограничивают лексический контекст (пространство имен): идентификаторы, определенные внутри такого блока, не видны за его пределами.

Идентификатор, определенный внутри данного пространства имен, перекрывает одноименный идентификатор, определенный вне этого пространства:

```
uint widgetCount;

void main() {
   writeln(widgetCount); // Выводит значение глобальной переменной
   auto widgetCount = getWidgetCount();
   writeln(widgetCount); // Выводит значение локальной переменной
}
```

При первом вызове функции `writeln` будет напечатано значение глобальной переменной `widgetCount`, при втором происходит обращение к локальной переменной `widgetCount`. Для доступа к глобальному идентификатору после того, как был определен перекрывающий его локальный идентификатор, служит точка, поставленная перед идентификатором (как уже говорилось в разделе 2.2.1), например `writeln(.widgetCount)`. Тем не менее запрещается определять идентификатор, если он перекрывает идентификатор, определенный в блоке верхнего уровня:

```
void main() {
   auto widgetCount = getWidgetCount();
   // Откроем вложенный блок
   {
      auto widgetCount = getWidgetCount(); // Ошибка!
   }
}
```

Если идентификаторы не перекрываются, то один и тот же идентификатор можно использовать внутри разных составных инструкций:

```
void main() {
   {
      auto i = 0;

   }
   {
      auto i = "eye"; // Без проблем

   }
   double i = 3.14;    // Тоже без проблем
}
```

Такой подход объясняется просто. Возможность перекрывать глобальные идентификаторы необходима, чтобы писать качественный модульный код, который собирается из нескольких отдельно скомпилированных частей; вы же не хотите, чтобы добавленная в локальное пространство имен глобальная переменная внезапно спутала все карты, запретив компиляцию невинных локальных переменных. С другой стороны, перекрытие локальных идентификаторов бесполезно с точки зрения модульности (поскольку в D составная инструкция никогда не простирается на несколько модулей) и обычно указывает либо на недосмотр (который вот-вот превратится в ошибку), либо на злокачественную функцию, вышедшую из-под контроля.

3.3. Инструкция if

Во многих примерах уже встречалась условная инструкция D `if`, которая очень похожа на то, чего вы могли от нее ожидать:

```
if (‹выражение›) ‹инструкция1›
```

или

```
if (‹выражение›) ‹инструкция1› else ‹инструкция2›
```

Достойна внимания одна деталь относительно инструкций, которыми управляет `if`. В отличие от других языков, в D нет «пустой инструкции», то есть отдельно точка с запятой сама по себе *не является* инструкцией и порождает ошибку. Это правило автоматически ограждает программистов от ошибок вроде следующей:

```
if (a == b);
   writeln("a и b равны");
```

В коротком коде подобная глупость очевидна, и вы легко ее устраните, но все меняется, когда выражение длиннее, сама инструкция затерялась

в дебрях кода, а на часах полвторого ночи. Если вы действительно хотите применить if к пустой инструкции, то можете использовать наиболее близкий аналог – пустую составную инструкцию:

```
if (a == b) {}
```

Это очень полезно, когда вы переделываете код, то и дело заключая фрагменты кода в комментарии и возвращая их обратно.

Часть условной инструкции с ключевым словом else всегда привязана к ближайшей части с ключевым словом if, так что отступы в следующем коде сделаны верно:

```
if (a == b)
   if (b == c)
      writeln("Все равны");
   else
      writeln("a не равно b. Но так ли это?");
```

Вторая функция writeln вызывается, когда a == b и b != c, потому что часть else привязана к внутреннему (второму) условию if. Если вы, напротив, хотите связать else с первым if, «буферизуйте» второе выражение с if с помощью пары фигурных скобок:

```
if ( a == b ) {
   if ( b == c )
      writeln("Все одинаковое");
   } else
   writeln("a отличается от b. Или это не так?");
```

Каскадные множественные конструкции if-else задаются в проверенном временем стиле C:

```
auto opt = getOption();
if (opt == "help") {

} else if (opt == "quiet") {

} else if (opt == "verbose") {

} else {
   stderr.writefln("Неизвестная опция '%s'"  opt);
}
```

3.4. Инструкция static if

Теперь, когда вы уже разогрелись на нескольких простых инструкциях (спасибо, что подавили этот зевок), можно взглянуть на нечто более необычное.

Если вы хотите «закомментировать» (или оставить) какие-то инструкции в зависимости от проверяемого во время компиляции логического условия, то вам пригодится инструкция `static if`[1]. Например[2]:

```
enum size_t
   g_maxDataSize = 100_000_000,
   g_maxMemory = 1_000_000_000;

double transmogrify(double x) {
   static if (g_maxMemory / 4 > g_maxDataSize) {
      alias double Numeric;
   } else {
      alias float Numeric;
   }
   Numeric[] y;
       // Сложные вычисления
   return y[0];
}
```

Инструкция `static if` позволяет осуществлять выбор во время компиляции и очень похожа на директиву `#if` языка C. Встречая `static if`, компилятор вычисляет условие. Если оно ненулевое, компилируется соответствующий код; иначе компилируется код, соответствующий выражению `else` (если таковое присутствует). В рассмотренном примере `static if` используется для переключения между экономичным (в отношении памяти) режимом работы (благодаря применению типа `float`, занимающего меньше места) и более точным режимом (благодаря применению более точного типа `double`). В обобщенном коде можно встретить и более мощные и выразительные примеры использования инструкции `static if`.

Выражение, проверяемое в `static if`, – это любое логическое выражение, которое можно вычислить во время компиляции. К разрешенным выражениям относится большое подмножество выражений языка, включая арифметические операции со значениями любых числовых типов, манипуляции с массивами, выражения `is` с типами в качестве аргументов (см. раздел 2.3.4.3) и даже вызовы функций (вычисление функций во время компиляции – действительно выдающееся средство). Вычисления во время компиляции подробно описаны в главе 5.

Срезание скобок

В примере с функцией `transmogrify` хорошо заметна одна странная особенность, а именно: тип `Numeric` определен внутри пары скобок { и }. Из-за этого он должен быть виден только локально, внутри пространства

---

[1] Да-да, это «еще одно место, где используется ключевое слово `static`».

[2] Тип `enum` будет рассмотрен позже. Для понимания примера надо знать, что значения объявленные как `enum`, определены на этапе компиляции, неизменны и могут использоваться в конструкциях, вычисляемых на этапе компиляции. – *Прим. науч. ред.*

имен, созданного при помощи этих скобок (и, следовательно, недоступен внутри включающей этот блок функции), что подрывает наш план на корню. Такое поведение также показало бы, насколько практически бесполезна многообещающая инструкция static if. Поэтому static if использует скобки для *группирования*, а не для *управления пространствами имен*. Там, где в фокус внимания попадают пространства имен и области видимости, static if срезает внешние скобки, если они есть (их необязательно ставить, если с помощью условной инструкции контролируется только одна инструкция; в нашем примере выше они используются только для того, чтобы не нарушить обязательство насчет стиля). Если вы действительно хотите поставить скобки (в их традиционном значении), просто добавьте еще одну пару:

```
import std.stdio;

void main() {
   static if (real.sizeof > double.sizeof) {{
      auto maximorum = real.max;
      writefln("Действительно большие числа - до %s!"  maximorum);
   }}
    .. /* maximorum здесь не виден */
}
```

Не только инструкция

Эта глава называется «Инструкции», а раздел — «Инструкция static if». Поэтому вы вправе немного удивиться, узнав, что static if — это не только инструкция, но и *объявление* (*declaration*). О «неинструкционности» static if свидетельствует не только срезание скобок, но и то, что static if может располагаться везде, где может быть расположено объявление, в том числе на недоступных инструкциям уровнях модулей, структур и классов. Например, мы можем определить Numeric глобально, просто вынеся соответствующий код за пределы функции transmogrify:

```
enum size_t
   g_maxDataSize = 100_000_000,
   g_maxMemory = 1_000_000_000;
..

// Объявление Numeric будет видно в контексте модуля
static if (g_maxMemory / 4 > g_maxDataSize) {
   alias double Numeric;
} else {
   alias float Numeric;
}

double transmogrify(double x) {
   Numeric[] y;
    .. // Сложные вычисления
   return y[0];
}
```

На два вида if – один else

У static if нет пары в виде static else. Вместо этого просто используется обычное ключевое слово else. В соответствии с логикой else привязывается к ближайшему if независимо от того, static if это или просто if:

```
if (a)
   static if (b) writeln("a и b ненулевые");
   else writeln("b равно нулю");
```

3.5. Инструкция switch

Лучше всего сразу проиллюстрировать работу инструкции switch примером:

```
import std.stdio;

void classify(char c) {
   write("Вы передали ");
   switch (c) {
      case '#':
         writeln("знак решетки.");
         break;
      case '0':    case '9':
         writeln("цифру.");
         break;
      case 'A':    case 'Z': case 'a':    case 'z':
         writeln("ASCII-знак.");
         break;
      case '.',    ':' ';'   '!'   '?':
         writeln("знак препинания.");
         break;
      default:
         writeln("всем знакам знак!");
         break;
   }
}
```

В общем виде инструкция switch выглядит так:

```
switch (‹выражение›) ‹инструкция›
```

‹выражение› может иметь числовой, перечисляемый или строковый тип; *‹инструкция›* может содержать метки (ярлыки, labels), определенные следующим образом:

1. case *‹в›*:

   Перейти сюда, если *‹выражение›* == *‹в›*. Чтобы можно было использовать внутри *в* запятые (см. раздел 2.3.18), все выражение требуется заключить в круглые скобки.

2. case ‹$в_1$›, ‹$в_2$›, ... ‹$в_n$›:

   Каждая запись вида ‹$в_k$› обозначает выражение. Рассматриваемая инструкция эквивалентна инструкции case ‹*элемент*$_1$›: case ‹*элемент*$_2$›: ... case ‹*элемент*$_n$›:.

3. case ‹$в_1$›: .. case ‹$в_2$›:

   Перейти сюда, если ‹*выражение*› >= ‹$в_1$› **и** ‹*выражение*› <= ‹$в_2$›.

4. default:

   Перейти сюда, если никакой другой переход невозможен.

‹*выражение*› **вычисляется один раз для всех этих проверок. Выражение в каждой метке** case **– это любое не противоречащее правилам языка выражение, которое можно проверить на равенство выражению** ‹*выражение*›**, а также на неравенство в случае использования синтаксиса с интервалом. Обычно** case**-выражения представлены константами, вычисляемыми во время компиляции, но D разрешает использовать и переменные, гарантируя, что вычисления будут производиться в порядке следования альтернатив до первого совпадения. По завершении вычислений выполняется переход к соответствующей метке** case **или** default **и выполнение программы продолжается из этой точки. Для того чтобы покинуть ветвление, используется инструкция** break**, осуществляющая выход из инструкции** switch**. В отличие от языков C и C++, D запрещает неявный переход к следующей метке и требует инструкции** break **или** return **после кода, соответствующего метке.**

```
switch (s) {
   case 'a': writeln("a"); // Вывести "a" и перейти к следующей метке
   case 'b': writeln("b");  // Ошибка! Неявный переход запрещен!
   default: break;
}
```

Если вы действительно хотите, чтобы после кода метки 'a' **выполнился код метки** 'b'**, вам придется явно указать это компилятору с помощью особой формы инструкции** goto**:**

```
switch (s) {
   case 'a': writeln("a"); goto case; // Вывести "a" и перейти
                                      // к следующей метке
   case 'b': writeln("b"); // После выполнения 'a' мы попадем сюда
   default: break;
}
```

Если же вы случайно забыли написать break **или** return**, компилятор любезно напомнит вам об этом. Можно было бы вообще отказаться от использования инструкции** break **в конструкции** switch**, но это нарушило бы обязательство компилировать C-подобный код по правилам языка C либо не компилировать его вообще.**

Для меток, вычисляемых во время компиляции, действует запрет: вычисленные значения не должны пересекаться. Пример некорректного кода:

```
switch (s) {
   case 'a' .. case 'z':     break;
   // Попытка задать особую обработку для 'w'
   case 'w':     break; // Ошибка! Case-метки не могут пересекаться!
   default: break;
}
```

Метка default должна быть обязательно объявлена. Если она не объявлена, компилятор сообщит об ошибке. Это сделано для того, чтобы предотвратить типичную для программистов ошибку – пропуск некоторого подмножества значений по недосмотру. Если такой опасности не существует, используйте default: break;, таким образом, аккуратно оформив ваше предположение. В следующем разделе описано, как статически гарантировать обработку всех возможных значений switch-условия.

3.6. Инструкция final switch

Инструкция switch обычно используется в связке с перечисляемым типом для обработки каждого из всех его возможных значений. Если во время эксплуатации число вариантов меняется, все зависимые переключатели неожиданно перестают соответствовать новому положению дел; каждую такую инструкцию необходимо вручную найти и изменить.

Теперь очевидно, что для получения масштабируемого решения следует заменить «переключение» на основе меток виртуальными функциями; в этом случае нет необходимости обрабатывать различные случаи в одном месте, но вместо этого обработка распределяется по разным реализациям интерфейса. Но в жизни не бывает все идеально: определение интерфейсов и классов требует серьезных усилий на начальном этапе работы над программой, чего можно избежать, остановившись на альтернативном решении с переключателем switch. В таких ситуациях может пригодиться инструкция final switch, статически «принуждающая» метки case покрывать все возможные значения перечисляемого типа:

```
enum DeviceStatus { ready, busy, fail }
 ..
void process(DeviceStatus status) {
   final switch (status) {
   case DeviceStatus.ready:
      ..
   case DeviceStatus.busy:

   case DeviceStatus.fail:

   }
}
```

Предположим, что при эксплуатации кода было добавлено еще одно возможное состояние устройства:

```
enum DeviceStatus { ready, busy, fail, initializing /* добавлено */ }
```

После этого изменения попытка перекомпилировать функцию process будет встречена отказом на следующем основании:

```
Error: final switch statement must handle all values
```
(Ошибка: инструкция final switch должна обрабатывать все значения)

Инструкция final switch требует, чтобы все значения типа enum были явно обработаны. Метки с интервалами вида case ‹$в_1$›: .. case ‹$в_2$›:, а также метку default: использовать запрещено.

3.7. Циклы

3.7.1. Инструкция while (цикл с предусловием)

Да, именно так:

```
while (‹выражение›) ‹инструкция›
```

Сначала вычисляется *‹выражение›*. Если оно ненулевое, выполняется *‹инструкция›* и цикл возобновляется: снова вычисляется *‹выражение›* и т. д. Иначе управление передается инструкции, расположенной сразу после цикла while.

3.7.2. Инструкция do-while (цикл с постусловием)

Если нужен цикл, который обязательно выполнится хотя бы раз, подойдет цикл с постусловием:

```
do ‹инструкция› while (‹выражение›);
```

Обратите внимание на обязательную точку с запятой в конце инструкции. Кроме того, после do должна быть хотя бы одна *‹инструкция›*. Цикл с постусловием эквивалентен циклу с предусловием, в котором сначала один раз выполняется *‹инструкция›*.

3.7.3. Инструкция for (цикл со счетчиком)

Синтаксис цикла со счетчиком:

```
for (‹определение счетчика› ‹выр1›; ‹выр2›) ‹инструкция›
```

Любое из выражений *‹определение счетчика›*, *‹выр$_1$›* и *‹выр$_2$›* (или все сразу) можно опустить. Если нет выражения *‹выр$_1$›*, считается, что оно истинно. Выражение *‹определение счетчика›* — это или объявление значения (например, auto i = 0; или float w;), или выражение с точкой с запятой в конце (например, i = 10;). По семантике использования цикл со счетчиком идентичен одноименным инструкциям из других языков: сначала вычисляется *‹определение счетчика›*, затем *‹выр$_1$›*; если оно истинно, выполняется *‹инструкция›*, потом вычисляется *‹выр$_2$›*, после чего выполнение цикла продолжается новым вычислением *‹выр$_1$›*.

3.7.4. Инструкция foreach (цикл просмотра)

Самое удобное, безопасное и зачастую быстрое средство просмотра значений в цикле – инструкция `foreach`[1], у которой есть несколько вариантов. Простейший вариант цикла просмотра:

```
foreach (‹идентификатор›; ‹выражение1›     ‹выражение2›) ‹инструкция›
```

‹*выражение*$_1$› и ‹*выражение*$_2$› могут быть числами или указателями. Попросту говоря, ‹*идентификатор*› проходит интервал от (включая) ‹*выражения*$_1$› до (не включая) ‹*выражения*$_2$›. Ради понятности этого неформального определения в нем не освещены некоторые детали. Например, сколько раз вычисляется ‹*выражение*$_2$› в процессе выполнения цикла – один или несколько? Или что происходит, если ‹*выражение*$_1$› >= ‹*выражение*$_2$›? Все это легко узнать, взглянув на семантически эквивалентный код, приведенный ниже. Техника представления высокоуровневых конструкций в терминах эквивалентных конструкций более простого (под)языка (в виде абстракций более низкого уровня) называется *снижением* (*lowering*). Она будет широко использоваться на протяжении всей этой главы.

```
{
   auto __n = ‹выражение2›;
   auto ‹идентификатор› = true ? ‹выражение1› : ‹выражение2›;
   for (; ‹идентификатор› < __n; ++‹идентификатор› ) ‹инструкция›
}
```

Здесь идентификатор `__n` генерируется компилятором, что гарантирует отсутствие конфликтов с другими идентификаторами[2] («свежее слово» в лексиконе тех, кто пишет компиляторы).

(Зачем нужны внешние фигурные скобки? Они гарантируют, что ‹*идентификатор*› не «просочится» за пределы цикла *foreach*, а также благодаря им вся эта конструкция – это одна инструкция.)

Теперь ясно, что и ‹*выражение*$_1$›, и ‹*выражение*$_2$› вычисляются всего один раз, а тип значения ‹*идентификатор*› определяется по правилам для тернарной условной операции (см. раздел 2.3.16) – вот в чем роль знаков ?:, никак не проявляющих себя при исполнении программы. Осторожное «примирение» типов, достигнутое благодаря знакам ?:, гарантирует предотвращение или, по крайней мере, выявление потенциальной неразберихи с числами разного размера и точности, а также конфликтов между типами со знаком и без знака.

Заметим, что компилятор принудительно не назначает `__n` какой-либо особый тип, то есть вы можете использовать этот вариант цикла `foreach` с пользовательскими типами, для которых определены оператор срав-

---

[1] Существует также цикл `foreach_reverse`, который работает аналогично `foreach`, но перебирает значения в обратном порядке.

[2] Идентификаторы, начинающиеся с двух подчеркиваний, описаны в разделе 2.1. – *Прим. пер.*

нения «меньше» (<) и оператор увеличения на единицу (мы научимся делать это в главе 12). Кроме того, если для типа не определен оператор <, но определен оператор сравнения на равенство, компилятор автоматически заменит оператор < оператором != при снижении. В этом случае не может быть проверена корректность задания интервала, поэтому вы должны удостовериться, что верхняя граница может быть достигнута с помощью повторного применения оператора ++, начиная с нижней границы. Иначе результат будет непредсказуемым.[1]

Заметим, что вы можете определить нужный тип счетчика внутри части *‹идентификатор›* определения цикла. Обычно такое объявление излишне, но полезно, если вы хотите, чтобы тип счетчика удовлетворял ряду особых условий, исключал путаницу в знаковости/беззнаковости или при необходимости использования неявного преобразования типов:

```
import std.math, std.stdio;

void main() {
   foreach (float elem; 1.0    100.0) {
      writeln(log(elem)); // Получает логарифм с одинарной точностью
   }
   foreach (double elem; 1.0    100.0) {
      writeln(log(elem)); // Двойная точность
   }
   foreach (elem; 1.0    100.0) {
      writeln(log(elem)); // То же самое
   }
}
```

3.7.5. Цикл просмотра для работы с массивами

Перейдем к другому варианту инструкции foreach, предназначенному для работы с массивами и срезами:

```
foreach (‹идентификатор›; ‹выражение›) ‹инструкция›
```

‹выражение› должно быть массивом (линейным или ассоциативным), срезом или иметь пользовательский тип. Последний случай мы рассмотрим в главе 12, а сейчас сосредоточимся на массивах и срезах. После того как *‹выражение›* было один раз вычислено, ссылка на него сохраняется в закрытой временной переменной. (Сам массив не копируется.) Затем переменной с именем *‹идентификатор›* по очереди присваивается каждый из элементов массива и выполняется *‹инструкция›*. Так же как

[1] В стандартной библиотеке (STL) C++ для определения завершения цикла последовательно используется оператор != на том основании, что (не)равенство – более общая форма сравнения, так как она применима к большему количеству типов. Подход D не менее общий, но при этом, когда это возможно, для повышения безопасности вычислений использует <, не проигрывая ни в обобщенности, ни в эффективности.

и в случае с циклом просмотра с интервалами, допускается указание типа перед *‹идентификатором›*.

Инструкция foreach предполагает, что во время итераций длина массива изменяться не будет; если вы задумали иное, возможно, вам стоит задействовать простой цикл просмотра и побольше внимания.

Обновление во время итерации

Присваивание переменной *‹идентификатор›* внутри *‹инструкции›* не отражается на состоянии массива. Если вы действительно хотите изменить элемент, рассматриваемый в текущей итерации, определите *‹идентификатор›* как ссылку, расположив перед ним ref или ref *‹тип›*. Например:

```
void scale(float[] array, float s) {
   foreach (ref e; array) {
      e *= s; // Обновляет массив "на месте"
   }
}
```

В приведенный код можно после ref добавить полное определение переменной e (включая ее тип), например ref float e. Однако на этот раз соответствие должно быть *точным*: ref запрещает преобразования типов!

```
float[] arr = [ 1.0, 2.5, 4.0 ];
foreach (ref float elem; arr) {
   elem *= 2; // Без проблем
}
foreach (ref double elem; arr) { // Ошибка!
   elem /= 2;
}
```

Причина такого поведения программы проста: чтобы гарантировать корректное присваивание, ref ожидает точного совпадения представления. Несмотря на то что из значения типа float всегда можно получить значение типа double, вы не вправе обновить значение типа float присваиванием типа double по нескольким причинам, самая очевидная из которых – разный размер этих типов.

Где я?

Иногда полезно иметь доступ к индексу итерации. Следующий вариант цикла просмотра позволяет привязать идентификатор к этому значению:

```
foreach (‹идентификатор1›, ‹идентификатор2›; ‹выражение›) ‹инструкция›
```

Так что можно написать:

```
void print(int[] array) {
   foreach (i, e; array) {
      writefln("array[%s] = %s;" i, e);
   }
}
```

Эта функция печатает содержание массива в виде, соответствующем коду на D. При выполнении `print([5, 2, 8])` выводится:

```
array[0] = 5;
array[1] = 2;
array[2] = 8;
```

Гораздо интереснее наблюдать за обращением к индексу элемента при работе с ассоциативными массивами:

```
void print(double[string] map) {
   foreach (i, e; map) {
      writefln("array['%s'] = %s;   i, e);
   }
}
```

Теперь `print(["Луна": 1.283, "Солнце": 499.307, "Проксима Центавра": 133814298.759])` выведет

```
array['Проксима Центавра'] = 1.33814e+08;
array['Солнце'] = 499.307;
array['Луна'] = 1.283;
```

Обратите внимание: элементы напечатаны не в том порядке, в каком они заданы в литерале. Интересно, что экспериментируя с тем же кодом, но в разных реализациях языка или разных версиях одной и той же реализации, можно наблюдать изменение порядка. Дело в том, что в ассоциативных массивах применяются таинственные методики, повышающие эффективность хранения и выборки элементов за счет отказа от гарантированного упорядочивания.

Тип индекса и самого элемента определяются по контексту. Можно действовать и по-другому, «навязывая» нужные типы одной из переменных *‹идентификатор$_1$›* и *‹идентификатор$_2$›* или обеим сразу. Однако помните, что *‹идентификатор$_1$›* не может быть ссылкой (перед ним нельзя поставить ключевое слово `ref`).

Проделки

Во время итерации можно по-разному изменять просматриваемый массив:

- *Изменение массива «на месте».* Во время итерации будут «видны» изменения еще не посещенных ячеек массива.
- *Изменение размера массива.* Цикл повторяется, пока не будет просмотрено столько элементов массива, сколько в нем было до входа в цикл. Возможно, в результате изменения размера массив будет перемещен в другую область памяти; в этом случае последующее изменение массива не видно во время итерации, а также последующие изменения, вызванные во время самой итерации, не отразятся на массиве. Использовать такую технику не рекомендуется, поскольку правила перемещения массива в памяти зависят от реализации.

- *Освобождение выделенной под массив памяти (полное или частичное; в последнем случае говорят о «сжатии» массива) с помощью низкоуровневых функций управления памятью.* Желая получить полный контроль и достичь максимальной эффективности, вы не пожалели времени на изучение низкоуровневого управления памятью по документации к своей реализации языка. Все, что можно предположить: 1) вы знаете, что творите, и 2) не скучно с вами лишь тому, кто написал собственный сборщик мусора.

3.7.6. Инструкции continue и break

Инструкция `continue` выполняет переход к началу новой итерации цикла, определяемого ближайшей к ней инструкцией `while`, `do-while`, `for` или `foreach`. Инструкции, расположенные между `continue` и концом тела цикла не выполняются.

Инструкция `break` выполняет переход к коду, расположенному сразу после ближайшей к ней инструкции `while`, `do-while`, `for`, `foreach`, `switch` или `final switch`, мгновенно завершая ее выполнение.

Обе инструкции можно использовать с необязательной меткой, указывающей, к какой именно инструкции они относятся. «Пометка» инструкций `continue` и `break` значительно упрощает построение сложных вариантов итераций, позволяя обойтись без переменных состояния и инструкции `goto`, описанной в следующем разделе.

```
void fun(string[] strings) {
   loop: foreach (s; strings) {
      switch (s) {
      default: ...; break;    // Выйти из инструкции switch
      case "ls":   .; break; // Выйти из инструкции switch
      case "rm": ...; break; // Выйти из инструкции switch
       ..
      case "#": break loop;  // Проигнорировать оставшиеся
                             // строки (прервать цикл foreach)
      }
   }

}
```

3.8. Инструкция goto (безусловный переход)

В связи с глобальным потеплением не будем горячиться по поводу инструкции `goto`. Достаточно сказать, что в D она имеет следующий синтаксис:

```
goto ‹метка›;
```

Идентификатор *‹метка›* должен быть виден внутри функции, где вызывается `goto`. Метка определяется явно как идентификатор с двоеточием, расположенный перед инструкцией. Например:

```
int a;

mylabel: a = 1;

if (a == 0) goto mylabel;
```

Нельзя переопределять метки внутри одной и той же функции. Другое ограничение состоит в том, что goto не может «перепрыгнуть» точку определения значения, видимого в точке «приземления». Например:

```
void main() {
   goto target;
   int x = 10;
   target: {} // Ошибка! goto заставляет пропустить определение x!
}
```

Наконец, инструкция goto не может выполнить переход за границу исключения (см. раздел 3.11). Таким образом, у инструкции goto почти нет ограничений, и именно это делает ее опасной. С помощью goto можно перейти куда угодно: вперед или назад, внутрь или за пределы инструкций if, внутрь и за пределы циклов, включая пресловутый переход прямо в середину тела цикла.

Тем не менее в D опасно не все, чего коснется goto. Если написать внутри конструкции switch инструкцию

```
goto case ‹выражение›;
```

то будет выполнен переход к соответствующей метке case *‹выражение›*. Инструкция

```
goto case;
```

выполняет переход к следующей метке case. Инструкция

```
goto default;
```

выполняет переход к метке default. Несмотря на то что эти переходы ничуть не более структурированы, чем любые другие случаи использования goto, их легче отслеживать, поскольку они расположены в одном месте программы и значительно упрощают структуру инструкции switch:

```
enum Pref { superFast, veryFast, fast, accurate,regular, slow, slower };
Pref preference;
double coarseness = 1;
...
switch (preference) {
   case Pref.fast: ...; break;
   case Pref.veryFast: coarseness = 1.5; goto case Pref.fast;
   case Pref.superFast: coarseness = 3; goto case Pref.fast;
   case Pref.accurate: ...; break;
   case Pref.regular: goto default;
   default:

}
```

При наличии инструкций break и continue с метками (см. раздел 3.7.6), исключений (см. раздел 3.11) и инструкции scope (мощное средство управления порядком выполнения программы, см. раздел 3.13) поклонникам goto все труднее найти себе достойное оправдание.

3.9. Инструкция with

Созданная по примеру Паскаля инструкция with облегчает работу с конкретным объектом.

Синтаксис:

```
with (‹выражение›) ‹инструкция›
```

Сначала вычисляется *‹выражение›*, после чего внутренние элементы верхнего уровня вложенности полученного объекта делаются видимыми внутри *‹инструкции›*. Мы уже встречались со структурами в главе 1, поэтому рассмотрим пример с применением типа struct:

```
import std.math, std.stdio;

struct Point {
   double x, y;
   double norm() { return sqrt(x * x + y * y); }
}
void main() {
   Point p;
   int z;
   with (p) {
      x = 3;           // Присваивает значение полю p.x
      p.y = 4;         // Хорошо, что все еще можно явно использовать p
      writeln(norm()); // Выводит значение поля p.norm, то есть 5
      z = 1;           // Поле z осталось видимым
   }
}
```

Изменения полей отражаются непосредственно на объекте, с которым работает инструкция with: она «распознает» в p l-значение и помнит об этом.

Если один из идентификаторов, включенный в область видимости с помощью инструкции with, перекрывает ранее определенный в функции идентификатор, то из-за возникшей неопределенности компилятор запрещает доступ к такому идентификатору. При том же определении структуры Point следующий код не скомпилируется:

```
void fun() {
   Point p;
   string y = "Я занимаюсь точкой (острю).";
   with (p)
      writeln(x, ":"  y); // Ошибка!
```

```
        // Полю p.y запрещено перекрывать переменную y!
    }
}
```

Однако об ошибке сообщается только в случае *реальной*, а не *потенциальной* неопределенности. Например, если бы инструкция `with` в последнем примере вообще не использовала идентификатор `y`, этот код скомпилировался бы и запустился, несмотря на скрытую неопределенность. Кроме того, программа работала бы при замене строки `writeln(x, ":", y);` строкой `writeln(x, ":", p.y);`, поскольку явное указание принадлежности идентификатора `y` объекту `p` полностью исключает неопределенность.

Инструкция `with` может перекрывать различные идентификаторы уровня модуля (то есть глобальные идентификаторы). Доступ к идентификаторам, перекрытым инструкцией `with`, осуществляется с помощью синтаксиса `.идентификатор`.

Заметим, что можно сделать неявной множественную вложенность объектов, написав:

```
with (‹выр1›) with (‹выр2›) .. with (‹вырn›) ‹инструкция›
```

При использовании вложенных инструкций `with` нет угрозы неопределенности, так как язык запрещает во внутренней инструкции `with` перекрывать идентификатор, определенный во внешней инструкции `with`. В двух словах: в D локальному идентификатору запрещено перекрывать другой локальный идентификатор.

3.10. Инструкция return

Чтобы немедленно вернуть значение из текущей функции, напишите

```
return ‹выражение›;
```

Эта инструкция вычисляет *‹выражение›* и возвращает полученное значение в точку вызова функции, предварительно неявно преобразовав его (если требуется) к типу, возвращаемому этой функцией.

Если текущая функция имеет тип `void`, *‹выражение›* должно быть опущено или представлять собой вызов функции, которая в свою очередь имеет тип `void`.

Выход из функции, возвращающей не `void`, должен осуществляться посредством инструкции `return`. Во время компиляции это трудно эффективно отследить, так что, возможно, иногда вы будете получать от компилятора необоснованные претензии.

3.11. Обработка исключительных ситуаций

Язык программирования D поддерживает обработку ошибок с помощью механизма исключительных ситуаций, или исключений (exceptions).

Исключение инициируется инструкцией throw, а обрабатывается инструкцией try. Чтобы породить исключение, обычно пишут:

```
throw new SomeException("Произошло нечто подозрительное");
```

Тип SomeException должен наследовать от встроенного класса Throwable. D не поддерживает создание исключительных ситуаций произвольных типов, отчасти потому, что, как мы скоро увидим, назначение определенного класса корневым облегчает обработку цепочек исключений разных типов.

Чтобы обработать исключение или просто быть в курсе, что оно произошло, используйте инструкцию try, которая в общем виде выглядит так:

```
try ‹инструкция›
catch (‹И1› ‹и1›) ‹инструкция1›
catch (‹И2› ‹и2›) ‹инструкция2›

catch (‹Иn› ‹иn›) ‹инструкцияn›
finally ‹инструкцияf›
```

Можно опустить компонент finally, как и любой из компонентов catch (или даже все компоненты catch). Однако должно соблюдаться условие: в инструкции try должен быть хотя бы один компонент finally или catch. ‹$И_к$› – это типы, которые, как уже сказано, должны наследовать от Throwable. Идентификаторы ‹$и_к$› связаны с захваченным объектом-исключением и могут отсутствовать.

Семантика всей инструкции такова. Сначала выполняется ‹*инструкция*›. Если при ее выполнении возникает исключение (назовем его ‹$и_х$› и будем считать, что оно имеет тип ‹$И_х$›), то предпринимаются попытки сопоставить типы ‹$И_1$›, ‹$И_2$›, ..., ‹$И_n$› с типом ‹$И_х$›. «Побеждает» первый тип ‹$И_к$›, который совпадает с ‹$И_х$› или является его предком. С объектом-исключением ‹$и_х$› связывается идентификатор ‹$и_к$› и выполняется ‹$инструкция_к$›. Исключение считается обработанным, поэтому если во время выполнения самой ‹$инструкции_к$› также возникнет исключение, информация о нем не будет разослана компонентам catch, содержащим возможные обработчики текущего исключения. Если ни один из типов ‹$И_к$› не подходит, исключение ‹$и_х$› всплывает по стеку вызовов с целью поиска обработчика.

Если компонент finally присутствует, ‹$инструкция_f$› выполняется абсолютно во всех случаях: независимо от того, порождается исключение или нет, и даже если исключение было обработано одним из компонентов catch и в результате было порождено новое исключение. Этот код гарантированно выполняется (если, конечно, не помешают бесконечные циклы и системные вызовы, вызывающие останов программы). Если и ‹$инструкция_f$› порождает исключение, оно будет присоединено к текущей цепочке исключений. Механизм исключений языка D подробно описан в главе 9.

Инструкция goto (см. раздел 3.8) не может совершить переход внутрь инструкций ‹*инструкция*›, ‹*инструкция*$_1$›, ..., ‹*инструкция*$_n$› и ‹*инструкция*$_f$›, кроме случая, когда goto находится внутри самой инструкции.

3.12. Инструкция mixin

Благодаря главе 2 (см. раздел 2.3.4.2) мы узнали, что с помощью выражений mixin можно преобразовывать строки, известные во время компиляции, в выражения на D, которые компилируются как обычный код. Инструкции с mixin предоставляют еще больше возможностей, позволяя создавать с помощью mixin не только выражения, но также объявления и инструкции.

Предположим, вы хотите как можно быстрее выяснить число ненулевых разрядов в байте. Это число, называемое весом Хемминга, используется в решении множества прикладных задач, таких как шифрование, распределенные вычисления и приближенный поиск в базе данных. Простейший способ подсчета ненулевых битов в байте: последовательно суммировать значения младшего бита, сдвигая каждый раз введенное число на один разряд вправо. Более быстрый метод был впервые предложен Питером Вегнером [60] и популяризирован Керниганом и Ричи в их классическом труде [34]:

```
uint bitsSet(uint value) {
   uint result;
   for (; value; ++result) {
      value &= value - 1;
   }
   return result;
}
unittest {
   assert(bitsSet(10) == 2);
   assert(bitsSet(0) == 0);
   assert(bitsSet(255) == 8);
}
```

Этот метод быстрее, чем самый очевидный, потому что цикл выполняется ровно столько раз, сколько ненулевых битов во введенном значении. Но функция bitsSet все равно тратит время на управление инструкциями; более быстрый метод – это обращение к нужной ячейке таблицы (в терминах D – к нужному элементу массива). Можно улучшить результат, заполнив таблицу еще во время компиляции; вот здесь и пригодится объявление с помощью инструкции mixin. Задумка в том, чтобы сначала создать строку, которая выглядит как объявление линейного массива, а затем с помощью mixin скомпилировать эту строку в обычный код. Генератор таблицы может выглядеть так:

```
import std.conv;

string makeHammingWeightsTable(string name, uint max = 255) {
```

```
    string result = "immutable ubyte[""to!string(max + 1)""] ""name"" = [ ";
    foreach (b; 0 .. max + 1) {
        result ~= to!string(bitsSet(b)) ~ "  ";
    }
    return result ~ "];";
}
```

Вызов функции makeHammingWeightsTable возвращает строку "immutable ubyte[256] t = [0, 1, 1, 2, ..., 7, 7, 8,];" Квалификатор immutable (см. главу 8) указывает, что таблица никогда не изменится после инициализации. С библиотечной функцией to!string мы впервые встретились в разделе 1.6. Эта функция преобразует в строку любое значение (в данном случае значения типа uint, возвращаемые функцией bitsSet). Теперь, когда у нас есть нужный код в виде строки, для определения таблицы достаточно выполнить всего одно действие:

```
mixin(makeHammingWeightsTable("hwTable"));
unittest {
    assert(hwTable[10] == 2);
    assert(hwTable[0] == 0);
    assert(hwTable[255] == 8);
}
```

Теоретически можно строить таблицы любого размера, но полученную программу всегда рекомендуется тестировать: из-за кэширования работа со слишком большими таблицами может на самом деле выполняться медленнее вычислений.

В качестве последнего средства (как тренер по айкидо скрепя сердце рекомендует ученикам газовый баллончик) стоит упомянуть сочетание импорта строки (инструкция import, см. раздел 2.2.5.1) и объявлений, созданных с помощью mixin, которое позволяет реализовать самую примитивную форму модульности – текстовое включение. Полюбуйтесь:

```
mixin(import("widget.d"));
```

Выражение import считывает текст файла widget.d в строковый литерал, который выражение mixin тут же преобразует в код. Используйте этот трюк, только если действительно считаете, что без него ваша честь хакера поставлена на карту.

3.13. Инструкция scope

Инструкция scope – нововведение D, хотя и другие языки в той или иной форме реализуют подобную функциональность. Инструкция scope позволяет легко писать на D корректно работающий код и, главное, без проблем читать и понимать его впоследствии. Можно и другими средствами достичь свойственной коду со scope корректности, однако, за исключением самых заурядных примеров, результат окажется непостижимым.

Синтаксис:

```
scope(exit) ‹инструкция›
```

‹инструкция› принудительно выполняется после того, как поток управления покинет текущую область видимости (контекст). Результат будет таким же, что и при использовании компонента `finally` инструкции `try`, но в общем случае инструкция `scope` более масштабируема. С помощью `scope(exit)` удобно гарантировать, что, оставляя контекст, вы «навели порядок». Допустим, в вашем приложении используется флаг `g_verbose` («говорливый»), который вы хотите временно отключить. Тогда можно написать:

```
bool g_verbose;
 ..
void silentFunction() {
   auto oldVerbose = g_verbose;
   scope(exit) g_verbose = oldVerbose;
   g_verbose = false;
   ..
}
```

Остаток кода «молчаливой» функции `silentFunction` может быть любым, с досрочными выходами и возможными исключениями, но вы можете быть полностью уверены, что по окончании ее выполнения, даже если наступит конец света или начнется потоп, флаг `g_verbose` будет корректно восстановлен.

Чтобы в общих чертах представить действие инструкции `scope(exit)`, определим для нее *снижение*, то есть общий метод преобразования кода, содержащего `scope(exit)`, в эквивалентный код с другими инструкциями, такими как `try`. Мы уже неформально применяли технику снижения, рассматривая работу цикла со счетчиком в терминах цикла с предусловием, а цикла просмотра – в терминах цикла со счетчиком.

Рассмотрим блок, содержащий инструкцию `scope(exit)`:

```
{
   ‹инструкции1›
   scope(exit) ‹инструкция2›
   ‹инструкции3›
}
```

Пусть явно отображенный вызов `scope` – первый в этом блоке, то есть *‹инструкции1›* не содержат вызовов `scope` (но инструкции *‹инструкция2›* и *‹инструкции3›* могут его содержать). Применив технику снижения, преобразуем этот код в код следующего вида:

```
{
   ‹инструкции1›
   try {
      ‹инструкции3›
   } finally {
```

```
        ‹инструкция2›
    }
}
```

На этом преобразование не заканчивается. Инструкции *‹инструкция$_2$›* и *‹инструкции$_3$›* также подвергаются снижению, поскольку могут содержать дополнительные инструкции scope. (Процесс снижения всегда конечен, так как фрагменты всегда строго меньше исходной последовательности.) Это означает, что код, содержащий несколько инструкций scope(exit), вполне корректен, даже в таких странных случаях, как scope(exit) scope(exit) scope(exit) writeln("?"). Посмотрим, что происходит в любопытном случае, когда в одном и том же блоке встречаются две инструкции scope(exit):

```
{
    ‹инструкции1›
    scope(exit) ‹инструкция2›
    ‹инструкции3›
    scope(exit) ‹инструкция4›
    ‹инструкции5›
}
```

Предположим, что ни одна из инструкций не содержит ни одного дополнительного вызова инструкции scope. Воспользовавшись снижением, получим:

```
{
    ‹инструкции1›
    try {
        ‹инструкции3›
        try {
            ‹инструкции5›
        } finally {
            ‹инструкция4›
        }
    } finally {
        ‹инструкция2›
    }
}
```

Этот громоздкий код поможет нам выяснить порядок выполнения нескольких инструкций scope(exit) в одном блоке. Проследив порядок выполнения инструкций в полученном коде, можно сделать вывод, что инструкция *‹инструкция$_4$›* выполняется до инструкции *‹инструкция$_2$›*. Обобщенно, инструкции scope(exit) выполняются по схеме LIFO[1]: в порядке, обратном их следованию в тексте программы.

Отслеживать порядок выполнения инструкций scope гораздо легче, чем порядок выполнения эквивалентного кода с конструкцией try/finally;

[1] LIFO – акроним «Last In – First Out» (последним пришел – первым ушел). – *Прим. пер.*

элементарно: инструкция scope гарантирует, что управляемая ею инструкция будет выполнена при выходе из контекста. Это позволяет вам защитить свой код от ошибок без неудобной иерархии конструкций try/finally – простым перечислением нужных действий в одной строке.

Предыдущий пример демонстрирует еще одно прекрасное свойство инструкции scope – масштабируемость. С учетом огромной масштабируемости эта инструкция просто неотразима. (В конце концов, если бы требовалось лишь изредка выполнять одну-единственную инструкцию scope, можно было бы вручную написать ее сниженный эквивалент по указанной выше методике.) Функциональность нескольких инструкций scope(exit) требует увеличения длины кода программы – при использовании самих инструкций scope(exit) и одновременного увеличения как длины, так и глубины кода – при использовании эквивалентных инструкций try. Причем в глубину код масштабируется очень слабо, к тому же приходится делить «владения» с другими составными инструкциями, такими как if или for. Еще один подходящий вариант масштабируемого решения – применение деструкторов в стиле C++ (также поддерживаемых D; см. главу 7), если только вам удастся снизить стоимость определения новых типов. Но если приходится определять класс только потому, что понадобился его деструктор (а зачем еще нужен класс типа CleanerUpper[1]?), то в плане масштабируемости это решение даже хуже вложенных инструкций try. Вкратце, если классы – вакуумная сварка, а инструкции try – жевательная резинка, то инструкция scope(exit) – эпоксидный суперклей.

Инструкция scope(success) *‹инструкция›* включает *‹инструкцию›* в «график» программы только в случае обычного выхода из текущей области видимости (не в результате исключения). Выполним снижение для инструкции scope(success). Код

```
{
   ‹инструкции1›
   scope(success) ‹инструкция2›
   ‹инструкции3›
}
```

превращается в

```
{
   ‹инструкции1›
   bool __succeeded = true;
   try {
      ‹инструкции3›
   } catch(Exception e) {
      __succeeded = false;
      throw e;
   } finally {
```

[1] CleanerUpper – «уборщик» (от англ. clean up – убирать, чистить). – *Прим. пер.*

```
        if (__succeeded) ‹инструкция2›
    }
}
```

Далее, инструкции *‹инструкция$_2$›* и *‹инструкции$_3$›* также подвергаются снижению; процесс повторяется, пока не останется вложенных инструкций scope.

Перейдем к более мрачному варианту инструкции scope – инструкции scope(failure) *‹инструкция›*. Такая запись предписывает выполнить *‹инструкцию›* только при выходе из текущего контекста в результате возникшей исключительной ситуации.

Снижение для инструкции scope(failure) практически идентично снижению для инструкции scope(success), с тем лишь отличием, что флаг __succeeded проверяется на равенство false, а не true. Код

```
{
    ‹инструкции1›
    scope(failure) ‹инструкция2›
    ‹инструкции3›
}
```

превращается в

```
{
    ‹инструкции1›
    bool __succeeded = true;
    try {
        ‹инструкции3›
    } catch(Exception e) {
        __succeeded = false;
        throw e;
    } finally {
        if (!__succeeded) ‹инструкция2›
    }
}
```

Далее выполняется снижение инструкций *‹инструкция$_2$›* и *‹инструкции$_3$›*.

Инструкция scope может пригодиться во многих ситуациях. Предположим, вы хотите создать файл способом транзакции – то есть не оставляя на диске «частично созданный» файл, если в процессе его создания произойдет сбой. Здесь можно поступить так:

```
import std.contracts, std.stdio;

void transactionalCreate(string filename) {
    string tempFilename = filename ~ ".fragment";
    scope(success) {
        std.file.rename(tempFilename, filename);
    }
    auto f = File(tempFilename, "w");
        // Спокойно пишете в f
}
```

Инструкция scope(success) заранее определяет цель работы функции. Эквивалентный код без scope получился бы гораздо более замысловатым; к тому же обычно программист слишком занят кодом, выполняющимся при ожидаемых условиях, чтобы найти время для обработки маловероятных ситуаций. Поэтому необходимо, чтобы язык максимально облегчал обработку ошибок.

Большой плюс такого стиля программирования состоит в том, что весь код обработки ошибок собран в начале функции transactionalCreate и никак не затрагивает основной код. При всей своей простоте функция transactionalCreate очень надежна: вы получаете или готовый файл, или временный файл-фрагмент, но только не «битый» файл, который кажется нормальным.

3.14. Инструкция synchronized

Инструкция synchronized имеет следующий синтаксис:

```
synchronized ( ‹выражение1›, ‹выражение2›, ... ) ‹инструкция›
```

С ее помощью можно расставлять контекстные блокировки в многопоточных программах. Семантика инструкции synchronized определена в главе 13.

3.15. Конструкция asm

D нарушил бы свой обет быть языком для системного программирования, если бы не предоставил некоторые средства для взаимодействия с ассемблером. И если вы любите трудности, то будете счастливы узнать, что в D есть тщательно определенный встроенный язык ассемблера для Intel *x*86. Кроме того, этот язык переносим между всеми реализациями D, работающими на машинах *x*86. Поскольку ассемблер зависит только от машины, а не от операционной системы, на первый взгляд это средство D не кажется революционным, тем не менее вы будете удивлены. Исторически сложилось, что каждая операционная система определяет собственный синтаксис ассемблера, не совместимый с другими ОС, поэтому, например, код, написанный для Windows, не будет работать под управлением Linux, так как синтаксисы ассемблеров этих операционных систем разительно отличаются друг от друга (что вряд ли оправданно). D разрубает этот гордиев узел, отказавшись от внешнего ассемблера, специфичного для каждой системы. Вместо этого компилятор сам выполняет синтаксический анализ и распознает инструкции ассемблерного языка. Чтобы написать код на ассемблере, делайте так:

```
asm ‹инструкция на ассемблере›
```

или так:

```
asm { ‹инструкции на ассемблере› }
```

Идентификаторы, видимые перед конструкцией asm, доступны и внутри нее: ассемблерный код может использовать сущности D. Ассемблер D описывается в главе 11; он покажется знакомым любому, кто работал с ассемблером *x*86. Всю информацию по ассемблеру D вы найдете в документации [12].

3.16. Итоги и справочник

D предоставляет все ожидаемые обычные инструкции, предлагая при этом и несколько новинок, таких как static if, final switch и scope. Таблица 3.1 – краткий справочник по всем инструкциям языка D (за подробностями обращайтесь к соответствующим разделам этой главы).

Таблица 3.1. Справочник по инструкциям (‹и› – инструкция, ‹в› – выражение, ‹о› – объявление, ‹х› – идентификатор)

Инструкция	Описание
‹в›;	Вычисляет ‹в›. Ничего не изменяющие выражения, включающие лишь встроенные типы и операторы, запрещены (см. раздел 3.1)
{‹$и_1$› ... ‹$и_2$›}	Выполняет инструкции от ‹$и_1$› до ‹$и_2$› по порядку, пока управление не будет явно передано в другую область видимости (например, инструкцией return) (см. раздел 3.2)
asm ‹и›	Машиннозависимый ассемблерный код (здесь ‹и› обозначает ассемблерный код, а не инструкцию на языке D). В настоящее время поддерживается ассемблер *x*86 с единым синтаксисом для всех поддерживаемых операционных систем (см. раздел 3.15)
break;	Прерывает выполнение текущей (ближайшей к ней) инструкции switch, for, foreach, while или do-while с переходом к инструкции, следующей сразу за ней (см. раздел 3.7.6)
break ‹х›;	Прерывает выполнение инструкции switch, for, foreach, while или do-while, имеющей метку ‹х›:, с переходом к инструкции, следующей сразу *за ней* (см. раздел 3.7.6).
continue;	Начинает новую итерацию текущего (ближайшего к ней) цикла for, foreach, while или do-while с пропуском оставшейся части этого цикла (см. раздел 3.7.6)
continue ‹х›;	Начинает новую итерацию цикла for, foreach, while или do-while, снабженного меткой ‹х›:, с пропуском оставшейся части этого цикла (см. раздел 3.7.6)

Инструкция	Описание
do ‹*и*› while (‹*в*›);	Выполняет ‹*и*› один раз и продолжает ее выполнять, пока ‹*в*› истинно (см. раздел 3.7.2)
for (‹$и_1$› ‹$в_1$›; ‹$в_2$›) ‹$и_2$›	Выполняет ‹$и_1$›, которая может быть инструкцией-выражением, определением значения или просто точкой с запятой, и пока ‹$в_1$› истинно, выполняет ‹$и_2$›, после чего вычисляет ‹$в_2$› (см. раздел 3.7.3)
foreach (‹*х*›; ‹$в_1$› .. ‹$в_2$›) ‹*и*›	Выполняет ‹*и*›, инициализируя переменную ‹*х*› значением ‹$в_1$› и затем последовательно увеличивая ее на 1, пока ‹*х*› < ‹$в_2$›. Цикл не выполняется, если ‹$в_1$› >= ‹$в_2$›. Как ‹$в_1$›, так и ‹$в_2$› вычисляются всего один раз (см. раздел 3.7.4)
foreach (ref$_{опц}$ ‹*х*›; ‹*в*›) ‹*и*›	Выполняет ‹*и*›, объявляя переменную ‹*х*› и привязывая ее к каждому из элементов ‹*в*› поочередно. Результатом вычисления ‹*в*› должен быть массив или любой пользовательский тип-диапазон (см. главу 12). Если присутствует ключевое слово ref, изменения ‹*х*› будут отражаться и на просматриваемой сущности (см. раздел 3.7.5)
foreach (‹$х_1$›, ref$_{опц}$ ‹$х_2$›; ‹*в*›) ‹*и*›	Аналогична предыдущей, но вводит дополнительное значение ‹$х_1$›. Если ‹*в*› – это ассоциативный массив, то ‹$х_1$› привязывается к ключу, а ‹$х_2$› – к рассматриваемому значению. Иначе ‹$х_1$› привязывается к целому числу, показывающему количество проходов цикла (начиная с 0) (см. раздел 3.7.5)
goto ‹*х*›;	Выполняет переход к метке ‹*х*›, которая должна быть определена в текущей функции как ‹*х*›: (см. раздел 3.8)
goto case;	Выполняет переход к следующей метке case текущей инструкции switch (см. раздел 3.8)
goto case ‹*х*›;	Выполняет переход к метке case ‹*х*› текущей инструкции switch (см. раздел 3.8)
goto default;	Выполняет переход к метке обработчика по умолчанию default текущей инструкции switch (см. раздел 3.8)
if (‹*в*›) ‹*и*›	Выполняет ‹*и*›, если ‹*в*› ненулевое (см. раздел 3.3)
if (‹*в*›) ‹$и_1$› else ‹$и_2$›	Выполняет ‹$и_1$›, если ‹*в*› ненулевое, иначе выполняет‹$и_2$›. Компонент else, расположенный в конце, относится к последней инструкции if или static if (см. раздел 3.3)

Таблица 3.1 (продолжение)

Инструкция	Описание
`static if (`‹*в*›`)`‹*о/и*›	Вычисляет ‹*в*› во время компиляции и, если ‹*в*› ненулевое, компилирует объявление или инструкцию ‹*о/и*›. Если объявление или инструкция ‹*о/и*› заключены в { и }, то одна пара таких скобок срезается (см. раздел 3.4)
`static if (`‹*в*›`)`‹$о/и_1$› `else` ‹$о/к_2$›	Аналогична предыдущей плюс в случае ложности ‹*в*› компилирует ‹$о/и_2$›. Часть `else`, расположенная в конце, относится к последней инструкции `if` или `static if` (см. раздел 3.4)
`return` ‹$в$›$_{опц}$`;`	Возврат из текущей функции. Возвращаемое значение должно быть таким, чтобы его можно было неявно преобразовать к объявленному возвращаемому типу. ‹*в*› может быть опущено, если возвращаемый тип функции – `void` (см. раздел 3.10)
`scope(exit)` ‹*и*›	Выполняет ‹*и*›, каким бы образом ни был осуществлен выход из текущего контекста (то есть с помощью `return`, из-за необработанной ошибки или по исключительной ситуации). Вложенные инструкции `scope` (в том числе с ключевыми словами `failure` и `success`, см. ниже) выполняются в порядке, обратном их определению в коде программы (см. раздел 3.13)
`scope(failure)` ‹*и*›	Выполняет ‹*и*›, если выход из текущего контекста осуществлен по исключительной ситуации (см. раздел 3.13)
`scope(success)` ‹*и*›	Выполняет ‹*и*› при нормальном выходе из текущего контекста (через `return` или по достижении конца контекста) (см. раздел 3.13)
`switch (`‹*в*›`)` ‹*и*›	Вычисляет ‹*в*› и выполняет переход к метке `case`, соответствующей ‹*в*› и расположенной внутри ‹*и*› (см. раздел 3.5)
`final switch (`‹*в*›`)` ‹*и*›	Аналогична предыдущей, но работает только с перечисляемыми значениями и во время компиляции проверяет, обработаны ли все возможные значения с помощью меток `case` (см. раздел 3.6)
`synchronized (`‹$в_1$›`,` ‹$в_2$›`...)`‹*и*›	Выполняет ‹*и*›, в то время как объекты, возвращаемые ‹$в_1$›, ‹$в_2$› и т. д., заблокированы. Выражения ‹$в_1$› должны возвращать объект типа `class` (см. раздел 3.14)

Инструкция	Описание
`throw (‹в›);`	Вычисляет ‹в› и порождает соответствующее исключение с переходом в ближайший подходящий обработчик `catch`. ‹в› должно иметь тип `Throwable` или наследующий от него (см. раздел 3.11)
`try ‹и› catch(‹$T_1$› ‹$x_1$›) ‹$и_1$› ... catch(‹$T_n$› ‹$x_n$›) ‹$и_n$› finally ‹$и_f$›`	Выполняет ‹и›. Если при этом возникает исключение, пытается сопоставить его тип с типами ‹T_1›, ..., ‹T_n› по порядку. Если k-е сопоставление оказалось удачным, то далее сопоставления не производятся и выполняется ‹$и_k$›. В любом случае (завершилось выполнение ‹и› исключением или нет) перед выходом из `try` выполняется ‹$и_f$›. Все компоненты `catch` и `finally` (но не то и другое одновременно) могут быть опущены (см. раздел 3.11)
`while (‹в›) ‹и›`	Выполняет ‹и›, пока ‹в› ненулевое (цикл не выполняется, если уже при первом вычислении ‹в› оказывается нулевым) (см. раздел 3.7.1)
`with (‹в›) ‹и›`	Вычисляет ‹в›, затем выполняет ‹и›, как если бы она была членом типа ‹в›: все используемые в ‹и› идентификаторы сначала ищутся в пространстве имен, определенном ‹в› (см. раздел 3.9)

4

Массивы, ассоциативные массивы и строки

Предыдущие главы лишь косвенно знакомили нас с массивами, ассоциативными массивами и строками (тут выражение, там литерал), пора уже познакомиться с ними по-настоящему. Оперируя только этими тремя типами данных, можно написать много хорошего кода, так что теперь, когда в нашем арсенале уже есть выражения и инструкции, самое время побольше узнать о массивах, ассоциативных массивах и строках.

4.1. Динамические массивы

Язык D предлагает очень простую, но гибкую абстракцию массивов. Для типа `T` справедливо, что `T[]` – это тип, представляющий собой непрерывную область памяти, содержащую элементы типа `T`. В терминах D `T[]` – это «массив значений типа `T`», или просто «массив значений `T`».

Динамический массив создается с помощью выражения `new` (см. раздел 2.3.6.1):

```
int[] array = new int[20]; // Создать массив для 20 целых чисел
```

Более простой и удобный вариант:

```
auto array = new int[20];  // Создать массив для 20 целых чисел
```

Все элементы только что созданного массива типа `T[]` инициализируются значением `T.init` (для целых чисел это `0`). После того как массив создан, для доступа к его элементам служит индексирующее выражение `array[n]`:

```
auto array = new int[20];
auto x = array[5]; // Корректны индексы от 0 до 19
```

```
assert(x == 0);     // Начальное значение для всех элементов массива:
                    // int.init = 0
array[7] = 42;      // Элементам массива можно присваивать значения
assert(array[7] == 42);
```

Число элементов, заданное в выражении new, не обязательно константа. Например, следующая программа создает массив случайной длины и заполняет его случайными числами, для генерации которых вызывает функцию uniform из модуля std.random:

```
import std.random;

void main() {
   // От 1 до 127 элементов
   auto array = new double[uniform(1, 128)];
   foreach (i; 0    array.length) {
      array[i] = uniform(0.0, 1.0);
   }

}
```

Цикл foreach можно переписать, чтобы обращаться непосредственно к каждому элементу массива, не используя индексы (см. раздел 3.7.5):

```
foreach (ref element; array) {
   element = uniform(0.0, 1.0);
}
```

Ключевое слово ref сообщает компилятору, что в нашем коде присваивания элементу element должны отражаться в исходном массиве. Иначе значения присваивались бы только копиям элементов массива.

Можно инициализировать массив особыми значениями (отличными от значений по умолчанию) с помощью литерала массива:

```
auto somePrimes = [ 2, 3, 5, 7, 11, 13, 17 ];
```

Еще один способ создать массив – дублировать существующий массив. При обращении к свойству .dup массива создается поэлементная копия этого массива:

```
auto array = new int[100];
..
auto copy = array.dup;
assert(array !is copy);    // Это разные массивы,
assert(array == copy);     // но с одинаковым содержимым
```

Наконец, если вы просто определите переменную типа T[], не инициализируя ее или инициализируя значением null, то получите «пустой массив» (null array). Пустой массив не имеет элементов, проверка на равенство такого массива константе null возвращает true.

```
string[] a;               // То же, что string[] a = null
assert(a is null);
```

```
assert(a == null);   // То же, что выше
a = new string[2];
assert(a !is null);
a = a[0    0];
assert(a !is null);
```

Благодаря последней строке этого кода обнаруживается нечто странное: пустой массив – это необязательно null.

4.1.1. Длина

Динамические массивы всегда «помнят» свою длину. Доступ к этому значению предоставляет свойство .length массива:

```
auto array = new short[55];
assert(array.length == 55);
```

Выражение array.length часто используется внутри индексирующего выражения для массива array. Например, обратиться к последнему элементу массива array можно с помощью выражения array[array.length - 1]. Чтобы упростить подобную запись, было разрешено внутри индексирующих выражений обозначать длину индексируемого массива идентификатором $.

```
auto array = new int[10];
array[9] = 42;
assert(array[$ - 1] == 42);
```

Изменение длины массива обсуждается в разделах 4.1.8–4.1.10.

4.1.2. Проверка границ

Что произойдет, если выполнить следующий код?

```
auto array = new int[10];
auto invalid = array[100];
```

Учитывая, что массивы всегда знают свою длину, можно легко вставить соответствующие проверки, так что вопрос о выполнимости задачи не стоит. Все дело в том, что проверка границ – это одна из тех вещей, которые ставят программиста перед мучительным выбором между быстродействием и безопасностью.

По соображениям безопасности необходимо постоянно проверять каким-либо способом корректность обращений к массиву. Доступ к памяти за пределами массива может привести к неконтролируемому поведению программы, сделать ее уязвимой для эксплойтов, вызывать сбои.

В то же время при текущей технологии компилирования полная проверка границ все еще сильно влияет на быстродействие. Эффективная проверка границ – тема для серьезного исследования. Популярный подход состоит в том, что сначала расставляют проверки везде, где выполняется обращение к массиву, а затем убирают те из них, которые

статический анализатор сочтет излишними. Обычно этот процесс быстро усложняется, особенно в тех случаях, когда при использовании массивов пересекаются границы процедур и модулей. Применяемые сегодня методы проверки границ требуют длительного анализа даже для скромных программ и позволяют избавиться лишь от части ненужных проверок [58].

В отношении головоломки с проверкой границ D находится между двух огней. Язык пытается одновременно предоставить как безопасность и удобство, свойственные современным языкам, так и предельное, ничем не ограниченное быстродействие, желательное для языка системного уровня. Проблема проверки границ подразумевает выбор между этими двумя крайностями, и D позволяет сделать этот выбор вам самим, вместо того чтобы сделать его за вас.

Во время компиляции D дважды делает выбор:

- между безопасным и системным модулями (см. раздел 11.2.2);
- между промежуточной (non-release) и итоговой (release) сборками (см. раздел 10.6).

D различает «безопасные» (safe) и «системные» (system) модули. Средний уровень безопасности – «доверенный» (trusted). Подразумеваются модули, которые предоставляют безопасный интерфейс, но могут осуществлять доступ системного уровня в рамках своей реализации. Выбор уровня доверенности написанных вами модулей – за вами. Во время компиляции безопасного модуля компилятор статически отключает все средства языка (включая непроверенную индексацию массивов), которые могут вызвать некорректный доступ к памяти. Компилируя системный или доверенный модуль, компилятор разрешает необработанный, непроверенный доступ к аппаратному обеспечению. Вы можете задать уровень определенной части модуля (безопасный, системный или доверенный), воспользовавшись специальной опцией командной строки или вставив атрибут:

```
@safe;
```

или

```
@trusted;
```

или

```
@system;
```

Выбранный уровень безопасности «действует» начиная с точки вставки соответствующего атрибута до следующей точки вставки или до конца файла (если больше нет вставленных атрибутов).

Механизм безопасности модулей подробно описан в главе 11, а сейчас главное из всей этой информации то, что вы как разработчик можете выбрать для своего модуля атрибут `@safe`, `@trusted` или `@system`.

Решение о выполнении *итоговой* сборки вашего приложения принимается независимо от безопасности модулей. Указать компилятору D собрать итоговую версию программы можно с помощью флага командной строки (`-release` в эталонной реализации). Для безопасного модуля границы проверяются *всегда*. Для системного модуля проверки границ вставляются только при *промежуточной* (не итоговой) сборке. При промежуточной сборке также вставляются и другие проверки, такие как выражения `assert` и проверки контрактов (последствия выбора итоговой сборки подробно обсуждаются в главе 10). Взаимосвязь между степенью безопасности модуля (безопасный/системный модуль) и режимом сборки (итоговая/промежуточная сборка) отражена в табл. 4.1.

Таблица 4.1. Проверка границ в зависимости от вида модуля и режима сборки

	Безопасный модуль	Системный модуль
Промежуточная сборка	✓	✓
Итоговая сборка (флаг `-release` для компилятора `dmd`)	✓	☠

Вас предупредили.

4.1.3. Срезы

Срезы – это мощное средство, позволяющее выбирать и использовать непрерывный фрагмент массива. Например, можно напечатать только вторую половину массива:

```
import std.stdio;

void main() {
   auto array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
   // Напечатать только вторую половину
   writeln(array[$ / 2    $]);
}
```

Эта программа напечатает:

```
5 6 7 8 9
```

Чтобы получить срез массива `array`, используйте форму записи `array[m .. n]` для выбора части массива, которая начинается элементом с индексом `m` и заканчивается элементом с индексом `n-1` (включая и этот элемент). Срез имеет тот же тип, что и сам массив, поэтому, например, можно присвоить срез тому же массиву, с которого сделан этот срез:

```
array = array[$ / 2    $];
```

В выражениях, обозначающих начало и конец среза, может участвовать идентификатор $, как и в случае обычной индексации, обозначающий

длину массива, срез которого требуется получить. Если m и n равны, это не является ошибкой: результатом в этом случае будет пустой срез. Нельзя задать m > n или n > array.length. Проверка таких «незаконных случаев» выполняется в соответствии с порядком, описанным в разделе 4.1.2.

Выражение array[0 .. $] получает срез, включающий все содержимое массива array. Это выражение встречается довольно часто, и тут язык помогает программистам, позволяя вместо записи array[0 .. $] использовать краткую форму array[].

4.1.4. Копирование

Объект массива содержит (или может почти мгновенно вычислить) как минимум два ключевых значения – верхнюю и нижнюю границы своих данных. Например, после выполнения кода

```
auto a = [1, 5, 2, 3, 6];
```

объект a окажется в состоянии, показанном на рис. 4.1. Массив «видит» только область, заключенную между его границами; заштрихованная область ему недоступна.

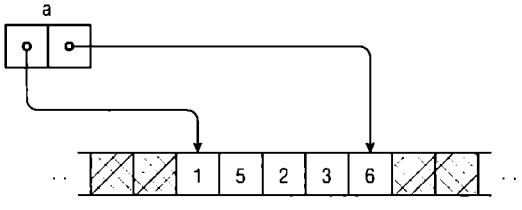

Рис. 4.1. Объект массива ссылается на область памяти, содержащую пять элементов

(Возможны и другие формы внутреннего представления массива, например, хранение адреса первого элемента и размера занимаемой области памяти или адреса первого элемента и адреса элемента, следующего за последним элементом. Тем не менее все представления в итоге предоставляют доступ к одной и той же существенной информации.)

Инициализация массива другим массивом (auto b = a), равно как и присваивание одного массива другому (int[] b; ... b = a;) не влечет скрытого автоматического копирования данных. Как показано на рис. 4.2, эти действия просто заставляют b ссылаться на ту же область памяти, что и a.

Более того, получение среза массива b сокращает область памяти, «видимую» b, также без всякого копирования b. При условии что исходное состояние массива задано на рис. 4.2, выполнение инструкции

```
b = b[1 .. $ - 2];
```

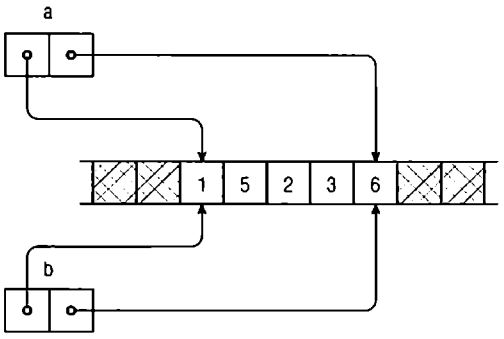

Рис. 4.2. *При выполнении инструкции* `auto b = a;` *содержимое* `a` *не копируется: вместо этого создается объект типа «массив», который ссылается на те же данные*

приведет лишь к сокращению диапазона, доступного `b`, без какого-либо копирования данных (рис. 4.3).

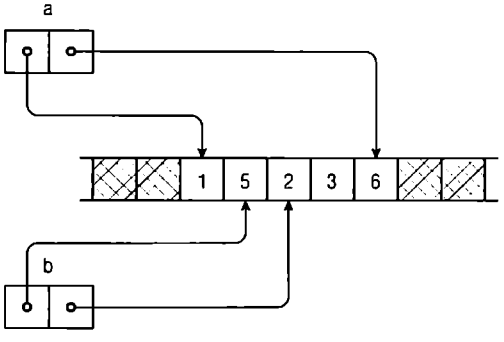

Рис. 4.3. *Выполнение инструкции* `b = b[1 .. $ - 2];` *ведет к сужению области памяти, управляемой переменной* `b`*, при этом выбранный срез не копируется*

Прямым следствием совместного использования (разделения) данных, проиллюстрированного рис. 4.2 и 4.3, является то, что изменение элемента одного массива может отразиться и на других массивах:

```
int[] array = [0, 1, 2];
int[] subarray = array[1 .. $];
assert(subarray.length == 2);
subarray[1] = 33;
assert(array[2] == 33); // Изменение массива subarray
                        // отразилось на массиве array
```

4.1.5. Проверка на равенство

Выражение `a is b` (см. раздел 2.3.4.3) сравнивает границы двух массивов на равенство и возвращает `true`, если и только если `a` и `b` привязаны в точности к одной и той же области памяти. Никакая проверка содержимого массивов не производится.

Для поэлементной проверки на равенство массивов `a` и `b` служит операция вида `a == b` или противоположная ей `a != b` (см. раздел 2.3.12).

```
auto a = ["hello", "world"];
auto b = a;
assert(a is b);  // Тест пройден, у a и b одни те же границы
assert(a == b);  // Естественно, тест пройден
b = a.dup;
assert(a == b);  // Тест пройден, a и b равны,
                 // хотя занимают разные области памяти
assert(a !is b); // Тест пройден, a и b различны, хотя
                 // имеют одинаковое содержимое
```

При поэлементном сравнении массивов просматриваются все элементы обоих массивов и соответствующие пары сравниваются по очереди с помощью оператора `==`.

4.1.6. Конкатенация

Конструкция

$$\text{содержимое}_1 \ \tilde{} \ \text{содержимое}_2$$

представляет собой выражение конкатенации. Результатом конкатенации является новый массив, содержимое которого представляет собой содержимое_1, за которым следует содержимое_2. Операндами в выражении конкатенации могут быть: два массива (типы `T[]` и `T[]`), массив и значение (типы `T[]` и `T`), значение и массив (типы `T` и `T[]`).

```
int[] a = [0, 10, 20];
int[] b = a ~ 42;
assert(b == [0, 10, 20, 42]);
a = b ~ a ~ 15;
assert(a.length == 8);
```

Под результирующий массив всегда выделяется новая область памяти.

4.1.7. Поэлементные операции

Некоторые операции применяются к массиву в целом, без явного указания на элементы массива. Чтобы применить поэлементную операцию, в выражении рядом с каждым срезом (в том числе слева от оператора присваивания) укажите `[]` или `[m .. n]`, как здесь:

```
auto a = [ 0.5, -0.5, 1.5, 2 ];
auto b = [ 3.5, 5.5, 4.5, -1 ];
```

```
auto c = new double[4];  // Память под массив должна быть уже выделена
c[] = (a[] + b[]) / 2;   // Рассчитать среднее арифметическое a и b
assert(c == [ 2.0, 2.5, 3.0, 0.5 ]);
```

В поэлементной операции могут участвовать:

- простое значение, например 5;
- срез, явно указанный с помощью [] или [m .. n], например a[] или a[1 .. $ - 1];
- любое корректное выражение на D с участием сущностей, определенных в двух предыдущих пунктах, унарных операторов - и ~, а также бинарных операторов +, -, *, /, %, ^^, ^, &, |, =, +=, -=, *=, /=, %=, ^=, &= и |=.

Поэлементная операция равносильна циклу, в котором поочередно каждому элементу массива, указанного слева от оператора присваивания, присваивается результат операции над элементами массивов с тем же индексом, расположенной справа от оператора присваивания. Например, присваивание

```
auto a = [1.0, 2.5, 3.6];
auto b = [4.5, 5.5, 1.4];
auto c = new double[3];
c[] += 4 * a[] + b[];
```

равносильно циклу

```
foreach (i; 0    c.length) {
   c[i] += 4 * a[i] + b[i];
}
```

Проверка границ выполняется в соответствии с порядком, описанным в разделе 4.1.2.

Используя явно заданные срезы (заканчивающиеся парой скобок [] или обозначением диапазона [m .. n]), числа и допустимые операторы, с помощью круглых скобок можно создавать выражения любой глубины и сложности, например:

```
double[] a, b, c;
double d;

a[] = -(b[] * (c[] + 4)) + c[] * d;
```

Из поэлементных операций чаще всего применяются простое заполнение ячеек (элементов) массива содержимым и их копирование:

```
int[] a = new int[128];
int[] b = new int[128];

b[] = -1;  // Заполнить все ячейки b значением -1
a[] = b[]; // Скопировать все данные из b в a
```

Предупреждение

Поэлементные операции очень мощны, а чем больше мощность, тем больше ответственность. Именно вы отвечаете за отсутствие перекрывания между l- и r-значениями каждого присваивания в поэлементной операции. Приводя высокоуровневые операции к примитивным операциям над векторами, которые может выполнять конечный процессор (на котором будет исполняться программа), компилятор вправе считать, что это именно так. Если вы намеренно используете перекрывание, то напишите циклы обработки элементов массива вручную, чтобы компилятор не смог выполнить какие-то непроверенные присваивания.

4.1.8. Сужение

Сужение массива означает, что массив должен «забыть» о некотором количестве своих начальных или конечных элементов без перемещения остальных. Ограничение на перемещение очень важно; если бы оно не было обязательным, массивы было бы легко сужать, просто создавая новую копию с теми элементами, которые требуется оставить.

Тем не менее сузить массив очень просто: нужно просто присвоить массиву срез его самого:

```
auto array = [0, 2, 4, 6, 8, 10];
array = array[0    $ - 2];        // Сужение справа на два элемента
assert(array == [0, 2, 4, 6]);
array = array[1    $];            // Сужение слева на один элемент
assert(array == [2, 4, 6]);
array = array[1    $ - 1];        // Сужение с обеих сторон
assert(array == [4]);
```

Все операции сужения выполняются за одно и то же время, которое не зависит от длины массива (практически они состоят из пары присваиваний с операндами-словами). Технически простое сужение массива с обоих концов – очень полезное средство языка D. (Другие языки позволяют легко сужать массивы справа, но не слева, поскольку такая операция повлекла бы перемещение всех элементов массива, чтобы сохранить положение левой границы массива.) В языке D вы можете получить копию массива и последовательно сужать ее, постоянно обрабатывая элементы в начале и в конце массива, в полной уверенности, что операции сужения, выполняемые за фиксированное время, не нанесут ощутимого ущерба быстродействию программы.

Напишем для примера маленькую программу, определяющую, является ли массив, переданный в командной строке, палиндромом. Массив-палиндром симметричен относительно своей середины, то есть `[5, 17, 8, 17, 5]` – это палиндром, а `[5, 7, 8, 7]` – нет. Для решения этой задачи нам потребуются несколько помощников. Сначала нужно извлечь аргументы командной строки в массив значений типа `string`. Эту задачу любезно возьмет на себя функция `main`, если определить ее как `main(string[] args)`.

Затем нужно преобразовать эти значения типа string в значения типа int, для чего мы воспользуемся функцией с говорящим именем to из модуля std.conv. Результат вычисления выражения to!int(str) – распознанное в строке str значение типа int. Все это помогает нам написать программу, которая проверяет, является ли введенный массив палиндромом:

```
import std.conv, std.stdio;

int main(string[] args) {
   // Избавиться от имени программы
   args = args[1 .. $];
   while (args.length >= 2) {
      if (to!int(args[0]) != to!int(args[$ - 1])) {
         writeln("не палиндром");
         return 1;
      }
      args = args[1    $ - 1];
   }
   writeln("палиндром");
   return 0;
}
```

Сначала программе нужно удалить свое имя из списка аргументов, формат которого соответствует традициям языка C. Если вызвать нашу программу (назовем ее palindrome) следующим образом:

```
palindrome 34 95 548
```

то содержимое массива args примет вид ["palindrome", "34", "95" "548"]. Вот где пригодилось сужение слева args = args[1 .. $]: оно сокращает массив args до массива ["34", "95", "548"]. Затем программа пошагово сравнивает элементы на концах массива. Если они не равны, то дальше можно не сравнивать: пишем "не палиндром" и закругляемся. А если проверка прошла успешно, то сужаем массив args с обоих концов. Только если все проверки возвратят true, а в массиве args останется не больше одного элемента (пустые массивы и массивы из одного элемента программа считает палиндромами), программа напечатает "палиндром" и завершится. Несмотря на то что программа активно манипулирует массивами, после инициализации массива args память не перераспределялась ни разу. Работа начинается с обращения к массиву args (память под который была выделена заранее), а потом он только сужается.

4.1.9. Расширение

Перейдем к расширению массивов. Расширить массив позволяет оператор присоединения ~=, например:

```
auto a = [87, 40, 10];
a ~= 42;
assert(a == [87, 40, 10, 42]);
```

```
a ~= [5, 17];
assert(a == [87, 40, 10, 42, 5, 17]);
```

У расширения массивов есть пара тонких моментов, связанных с перераспределением памяти. Рассмотрим код:

```
auto a = [87, 40, 10, 2];
auto b = a;          // Теперь a и b ссылаются на одну и ту же область памяти
a ~= [5, 17];        // Присоединить к a
a[0] = 15;           // Изменить a[0]
assert(b[0] == 15); // Будет ли пройден тест?
```

Повлияет ли выполненное после присоединения присваивание элементу a[0] на b[0]? Другими словами, будут ли a и b разделять данные после перераспределения памяти? Коротко на этот вопрос можно ответить так: b[0] может содержать 15, а может и не содержать – язык не дает никаких гарантий.

Реальность такова, что в конце массива a не всегда достаточно места, чтобы перераспределить память под измененный массив в том же месте. Иногда перенос массива в другую область памяти бывает неизбежен. Проще всего добиться корректного поведения программы в подобных случаях, *всегда* перераспределяя память под массив a после присоединения к нему новых элементов с помощью оператора ~=, то есть делая операцию a ~= b тождественной операции a = a ~ b, что означает: «Определить в новой области памяти массив, содержащий последовательность элементов массива a, к которой присоединена последовательность элементов массива b, и связать переменную a с полученным новым массивом». Такое поведение проще всего реализуется, но наносит серьезный урон быстродействию. Приведем пример. Обычно содержимое массивов пошагово наращивают в цикле:

```
int[] a;
foreach (i; 0 .. 100) {
   a ~= i;
}
```

При 100 элементах приемлема любая стратегия расширения и неважно, какой вариант будет выбран, но с ростом массивов только жесткие решения останутся относительно быстрыми. Не очень привлекательный подход состоит в том, чтобы разрешить удобный, но неэффективный синтаксис расширения a ~= b и применять его только с короткими массивами, а для длинных массивов использовать другой, менее удобный синтаксис. Маловероятно, что самый простой и интуитивно понятный синтаксис сработает как с короткими, так и с длинными массивами.

D оставляет оператору ~= свободу перераспределять память по своему усмотрению: он может выполнять расширение с переносом массива в новую область памяти, но старается оставить его на «старом месте», если после массива достаточно свободной памяти для размещения новых элементов. Выбор в пользу той или иной альтернативы определяется

исключительно реализацией ~=, но с той гарантией, что программа, выполняющая много присоединений к одному и тому же массиву, будет обладать хорошим *средним* быстродействием.

На рис. 4.4 показаны два возможных исхода расширения a ~= [5, 17].

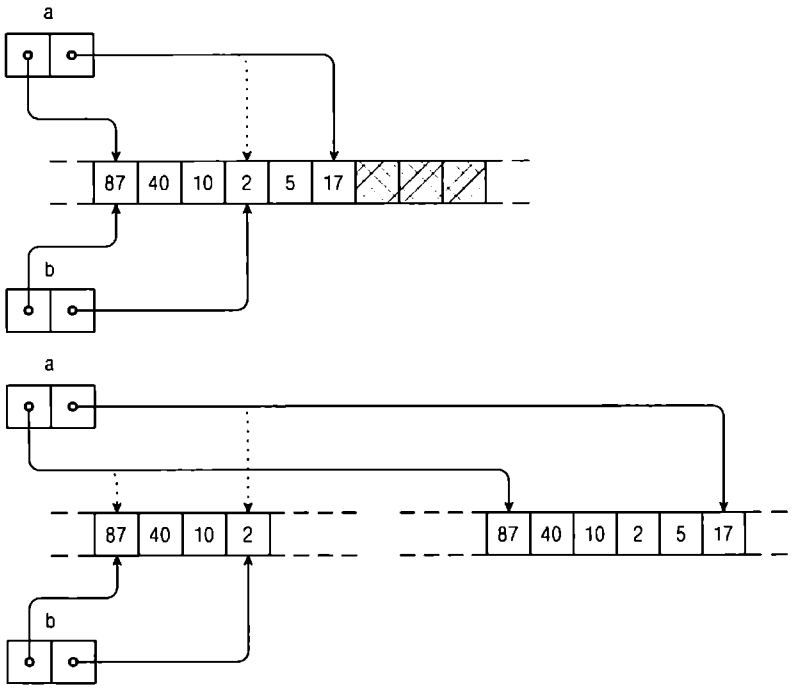

***Рис. 4.4.** Два возможных исхода попытки расширить массив a*

В зависимости от того, как работает низкоуровневый менеджер памяти, массив может расширяться разными способами:

- Обычно менеджеры памяти выделяют память только фиксированными блоками (то есть блоками размера, кратного 2). Поэтому возможно, что при запросе 700 байт будет выделено 1024 байта памяти, из которых 324 будут пустовать. Получив запрос на расширение, массив может проверить, нет ли такой незанятой памяти, и использовать ее.
- Если собственной незанятой памяти не осталось, массив может затеять более сложные переговоры с низкоуровневым менеджером памяти: «Послушай, мне бы немного памяти на благое дело. Нет ли случайно рядом со мной свободного блока?» Если менеджер памяти обнаружит незанятый блок справа от текущего блока массива, то объединит их. Такая операция называется *слиянием* (*coalescing*). После

этого расширение может продолжаться без перемещения каких-либо данных.

- Наконец, если справа от текущего блока совсем нет места, менеджер памяти выделяет новый блок памяти, и все содержимое массива копируется туда. Реализация менеджера памяти может принудительно резервировать дополнительную область памяти, например, обнаружив повторяющиеся расширения одного и того же массива.

Расширяющийся массив никогда не «наступит» на существующий массив. Например:

```
int[] a = [0, 10, 20, 30, 40, 50, 60, 70];
auto b = a[4    $];
a = a[0 .. 4];
// Сейчас a и b примыкают друг к другу
a ~= [0, 0, 0, 0];
assert(b == [40, 50, 60, 70]); // Тест пройден; массив a был перенесен
                               // в новую область памяти
```

Этот код искусно заставляет массив a думать, что в конце области памяти, которую он занимает, есть свободное место: первоначально массив a был больше по размеру, затем массив b занял вторую половину массива a, а сам массив a сузился до своей первой половины. Перед добавлением новых элементов в массив a массивы a и b занимали соседние области памяти: массив a находился слева от массива b. Однако успешное выполнение теста assert после добавления новых элементов в массив a подтвердило, что этот массив был перенесен в другую область памяти, а не расширился на том же месте. Оператор расширения добавляет в массив элементы без изменения адреса массива, только если уверен, что справа от расширяющегося массива нет другого массива, и при малейшем сомнении всегда готов подстраховаться, перераспределив память.

4.1.10. Присваивание значения свойству .length

Присвоив значение свойству .length массива, вы можете сузить или расширить его, в зависимости от отношения новой длины к старой. Например:

```
int[] array;
assert(array.length == 0);
array.length = 1000;            // Расширяется
assert(array.length == 1000);
array.length = 500;             // Сужается
assert(array.length == 500);
```

Если массив расширяется в результате присваивания свойству .length, добавленные элементы инициализируются значением T.init. Стратегия расширения и гарантии идентичности в этом случае аналогичны добавлению элементов с помощью оператора ~= (см. раздел 4.1.9).

Если массив сжимается в результате присваивания свойству .length, D гарантирует, что массив не будет перемещен. Практически, если n <= a.length, a.length = n эквивалентно a = a[0 .. n]. (Однако нет гарантии, что массив не будет перемещен в результате последующих расширений.)

Можно одновременно выполнить чтение, изменение и запись значения свойства .length следующим способом:

```
auto array = new int[10];
array.length += 1000;          // Расширяется
assert(array.length == 1010);
array.length /= 10;            // Сужается
assert(array.length == 101);
```

Здесь нет никакой магии; все, что необходимо сделать компилятору, — это переписать выражение array.length ‹o›= b в несколько иной форме: array.length = array.length ‹o› b. И все-таки немного магии тут есть (на самом деле, всего лишь ловкость рук): в переписанном выражении массив вычисляется всего лишь раз, что очень кстати, если реально array — это какое-то замысловатое выражение.

4.2. Массивы фиксированной длины

D также позволяет создать массив, длина которого известна во время компиляции. Пример объявления такого массива:

```
int[128] someInts;
```

Каждое сочетание типа T и размера n представляет собой уникальный тип T[n]: например, тип uint[10] отличается от типа uint[11], равно как и от типа int[10].

Память под все массивы фиксированной длины выделяется статически, в месте их объявления. Если значение массива определено глобально, память под него выделяется в сегменте данных программы, индивидуальном для каждого потока. Если же массив определяется внутри функции, он будет размещен в стеке этой функции при ее вызове. (Это означает, что определять слишком большие массивы внутри функций довольно опасно.) Однако если задать массив внутри функции с ключевым словом static, он займет блок памяти в сегменте данных потока, так что в этом случае риска переполнения стека нет.

При создании массива фиксированной длины типа T[n] все его элементы инициализируются значением T.init. Например:

```
int[3] a;
assert(a == [0, 0, 0]);
```

Также можно инициализировать массив типа T[n] с помощью литерала:

```
int[3] a = [1, 2, 3];
assert(a == [1, 2, 3]);
```

Но будьте осторожны: если в объявлении типа заменить `int[3]` ключевым словом `auto`, то по принятым в D правилам определения типов массиву `a` будет присвоен тип `int[]`, а не `int[3]`. Несмотря на то что кажется логичным выбрать тип `int[3]`, в некотором смысле более «точный», чем `int[]`, на практике динамические массивы используются гораздо чаще массивов фиксированной длины, поэтому трактовка литералов массивов как массивов фиксированной длины отрицательно сказалась бы на удобстве языка, став источником многих неприятных сюрпризов. Кроме того, такое толкование литералов свело бы на нет смысл использования ключевого слова `auto` с массивами. Поэтому значениям, задаваемым литералом, `T[]` присваивается по умолчанию, а `T[n]` – если вы *просите* присвоить этот конкретный тип и при этом `n` соответствует числу значений в литерале (как в коде выше).

Если вы инициализируете массив фиксированной длины типа `T[n]` с помощью единственного значения типа `T`, все ячейки массива будут заполнены этим значением.

```
int[4] a = -1;
assert(a == [-1, -1, -1, -1]);
```

Если вы планируете оставить массив неинициализированным и заполнить его во время исполнения программы, просто укажите в качестве инициализирующего значения ключевое слово `void`:

```
int[1024] a = void;
```

Возможность выделять память под массив, не инициализируя ее, особенно полезна, когда требуется задать большой массив под временный буфер. Будьте осторожны: неинициализированное целое число, скорее всего, никому особо не навредит, а вот неинициализированные значения ссылочных типов (таких как многомерные массивы) небезопасны. Доступ к элементам массива фиксированной длины осуществляется по индексу `a[i]`, как и к элементам динамических массивов. Просмотр массива фиксированной длины также практически идентичен просмотру динамического массива. Например, так создается массив, содержащий 1024 случайных числа:

```
import std.random;

void main() {
   double[1024] array;
   foreach (i; 0    array.length) {
      array[i] = uniform(0.0, 1.0);
   }

}
```

В цикле можно не использовать индекс, осуществляя доступ к элементу массива по ссылке:

```
foreach (ref element; array) {
   element = uniform(0.0, 1.0);
}
```

4.2.1. Длина

Очевидно, что массив фиксированной длины знает свой размер, потому что он «прошит» в его типе. В отличие от длины динамических массивов, свойство .length массива фиксированной длины неизменяемо и является статической константой. Это означает, что вы можете использовать это свойство везде, где требуется значение, известное во время компиляции, например, в качестве размера другого массива фиксированной длины при его определении:

```
int[100] quadrupeds[1];
int[4 * quadrupeds.length] legs; // Все в порядке, 400 ног
```

В индексирующем выражении массива a вместо записи a.length можно использовать идентификатор $, и значение этого выражения также будет известно во время компиляции.

4.2.2. Проверка границ

Проверка границ массивов фиксированной длины имеет интересную особенность. Если индексирование осуществляется с помощью выражения, вычисляемого во время компиляции, компилятор всегда проверяет, корректно ли оно, и отказывается компилировать программу, если обнаруживает попытку доступа к памяти за пределами массива. Например:

```
int[10] array;
array[15] = 5; // Ошибка!
// Индекс 15 находится за пределами a[0    10]!
```

Однако если индексирующее выражение вычисляется в процессе исполнения программы, проверка границ во время компиляции осуществляется настолько, насколько это возможно, а проверка границ во время исполнения программы делается по тем же правилам, что и проверка границ динамических массивов (см. раздел 4.1.2).

4.2.3. Получение срезов

Получение среза массива типа T[n] порождает массив типа T[] без копирования данных:

```
int[5] array = [40, 30, 20, 10, 0];
auto slice1 = array[2 .. $];        // slice1 имеет тип int[]
assert(slice1 == [20, 10, 0]);
auto slice2 = array[];              // Такой же, как array[0    $]
assert(slice2 == array);
```

[1] quadrupeds (англ.) – четвероногие. – *Прим. пер.*

Проверка границ во время компиляции выполняется для одной из границ массива или для обеих границ, если они известны на этом этапе.

Если вы примените к массиву типа `T[n]` оператор среза, указав в качестве границ среза числа `a1` и `a2`, известные во время компиляции, и при этом *укажете*, что должен быть возвращен массив типа `T[a2 - a1]`, компилятор удовлетворит ваш запрос. (По умолчанию, то есть при наличии ключевого слова `auto`, возвращается тип среза `T[]`.) Например:

```
int[10] a;
int[] b = a[1    7];  // Все в порядке
auto c = a[1    7];   // Все в порядке, с также имеет тип int[]
int[6] d = a[1    7]; // Все в порядке, срез a[1    7] скопирован в d
```

4.2.4. Копирование и неявные преобразования

В отличие от динамических массивов, массивы фиксированной длины копируются по значению. Это означает, что при копировании массива, передаче его внутрь функции в качестве аргументов и возврате из функции копируется весь массив. Например:

```
int[3] a = [1, 2, 3];
int[3] b = a;
a[1] = 42;
assert(b[1] == 2);          // b - независимая копия a
int[3] fun(int[3] x, int[3] y) {
   // x и y - копии переданных аргументов
   x[0] = y[0] = 100;
   return x;
}
auto c = fun(a, b);         // c имеет тип int[3]
assert(c == [100, 42, 3]);
assert(b == [1, 2, 3]);     // Вызов fun никак на отразился на b
```

Передача целых массивов по значению может быть неэффективной в случае большого массива, но у такого способа много преимуществ. Одно из них в том, что короткие массивы и передача по значению часто используются в высокопроизводительных вычислениях. Другое – в том, что от передачи по значению есть простое средство: когда бы вы ни пожелали передать массив по ссылке, просто используйте ключевое слово `ref` или автоматическое приведение к типу `T[]` (см. следующий абзац). Наконец, передача по значению делает работу с массивами фиксированной длины более согласованной с другими аспектами языка. (Раньше в D массивы фиксированной длины копировались по ссылке, но при такой семантике копирования многие случаи требуют особой обработки, что нарушает логику пользовательского кода.)

Массив типа `T[n]` может быть неявно преобразован к типу `T[]`. Память под динамический массив, полученный таким способом, не выделяется заново: он просто привязывается к границам исходного массива. Поэтому преобразование считается небезопасным, если исходный массив

расположен в стеке. Неявное преобразование типов облегчает передачу массивов фиксированной длины типа T[n] в функции, ожидающие значение типа T[]. Тем не менее, если функция возвращает значение типа T[n], результат ее вызова не может быть автоматически преобразован к типу T[].

```
double[3] point = [0, 0, 0];
double[] test = point;          // Все в порядке
double[3] fun(double[] x) {
   double[3] result;
   result[] = 2 * x[];          // Операция над массивом в целом
   return result;
}
auto r = fun(point);            // Все в порядке, теперь r имеет тип double[3]
```

Свойство .dup позволяет получить дубликат массива фиксированной длины (см. раздел 4.1), но вы получите не объект типа T[n], а динамически выделенный массив типа T[], содержащий копию массива фиксированной длины. Такое поведение оправданно, ведь чтобы получить копию статического массива a того же типа, не нужно прибегать ни к каким дополнительным ухищрениям – просто напишите auto copy = a.

4.2.5. Проверка на равенство

Массивы фиксированной длины можно проверять на равенство с помощью операторов is и ==, как и динамические массивы (см. раздел 4.1.5). Также можно смело использовать в проверках одновременно массивы обоих видов:

```
int[4] fixed = [1, 2, 3, 4];
auto anotherFixed = fixed;
assert(anotherFixed !is fixed); // Не то же самое (копирование по значению)
assert(anotherFixed == fixed);  // Те же данные
auto dynamic = fixed[];         // Получает границы массива fixed
assert(dynamic is fixed);
assert(dynamic == fixed);       // Естественно
dynamic = dynamic.dup;          // Создает копию
assert(dynamic !is fixed);
assert(dynamic == fixed);
```

4.2.6. Конкатенация

Конкатенация выполняется по тем же правилам, что и для динамических массивов (см. раздел 4.1.6). Нужно лишь помнить важную деталь. Вы получите массив фиксированной длины, только если *явно запросите* массив фиксированной длины. Иначе вы получите заново выделенный динамический массив. Например:

```
double[2] a;
double[] b = a ~ 0.5;   // Присоединить к double[2] значение,
                        // получить double[]
```

```
auto c = a ~ 0.5;          // То же самое
double[3] d = a ~ 1.5;     // Все в порядке, явный запрос
                           // массива фиксированной длины
double[5] e = a ~ d;       // Все в порядке, явный запрос
                           // массива фиксированной длины
```

Если в качестве результата конкатенации ~ явно указан массив фиксированной длины, никогда не происходит динамического выделения памяти: статически выделяется блок памяти, и результат конкатенации копируется в него.

4.2.7. Поэлементные операции

Поэлементные операции с массивами фиксированной длины работают так же, как и одноименные операции для динамических массивов (см. раздел 4.1.7). Компилятор всегда старается проверить корректность доступа к элементам массивов фиксированной длины в поэлементных выражениях.

4.3. Многомерные массивы

Поскольку запись T[] означает динамический массив элементов типа T, а T[], в свою очередь, – тоже тип, легко сделать вывод, что T[][] – это массив элементов типа T[], то есть массив массивов элементов типа T. Каждый элемент «внешнего» массива – это, в свою очередь, тоже массив, предоставляющий обычную функциональность, присущую массивам. Рассмотрим T[][] на практике:

```
auto array = new double[][5]; // Массив из пяти массивов, содержащих
                              // элементы типа double, первоначально
                              // каждый из них - null
// Сделать треугольную матрицу
foreach (i, ref e; array) {
   e = new double[array.length   i];
}
```

Здесь определен массив треугольной формы: первая строка содержит пять элементов типа double, вторая – четыре и так далее до последней строки (с номером 4), в которой всего один элемент. Многомерный массив, полученный простым составлением из динамических массивов, называют *зубчатым массивом* (*jagged array*), поскольку его строки могут иметь разную длину (в отличие от массива с ровным правым краем, содержащего строки одинаковой длины). На рис. 4.5 показано расположение массива array в памяти.

Чтобы получить доступ к элементу зубчатого массива, поочередно укажите индексы для каждого измерения, например array[3][1] – обращение ко второму элементу четвертой строки зубчатого массива.

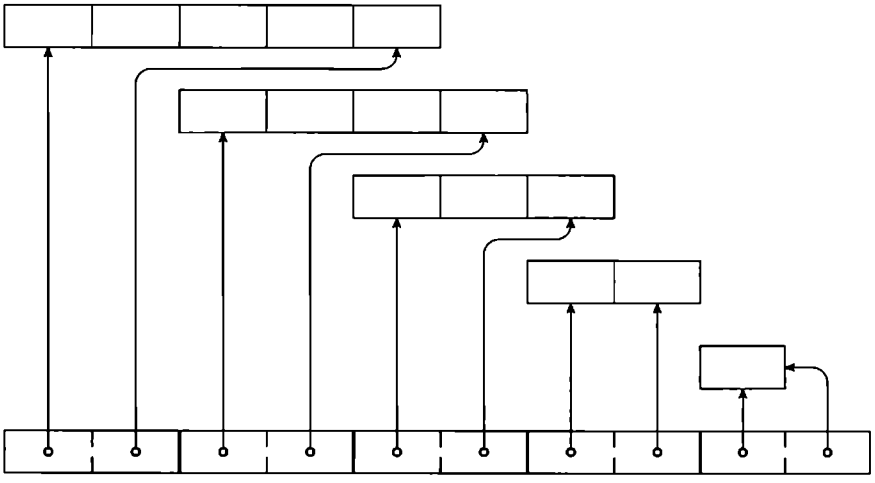

Рис. 4.5. Зубчатый массив из примера, содержащий треугольную матрицу

Зубчатые массивы не являются непрерывными. Плюс этого свойства в том, что такой массив может быть рассредоточен по разным областям памяти и не требует слишком большого непрерывного блока. Кроме того, возможность хранить строки разной длины позволяет хорошо экономить память. Минусом же является то, что «высокий и худой» массив с большим числом строк и малым числом столбцов требует больших накладных расходов, поскольку для хранения содержимого каждого столбца требуется массив. Например, массив из 1000000 строк, в каждом из которых всего по 10 значений типа `int`, занимает 2000000 слов (одна строка – один массив) плюс дополнительные расходы на неиспользованную память при выделении 1000000 маленьких блоков, что, в зависимости от реализации менеджера памяти, может оказаться ощутимо гораздо больше затрат на хранение содержимого каждой строки (на каждые 10 целых чисел нужно всего по 40 байт).

При работе с зубчатыми массивами могут возникнуть проблемы со скоростью доступа и дружелюбностью кэша. Каждое обращение к элементам такого массива – это на самом деле два косвенных обращения: на первом шаге через внешний массив осуществляется доступ к нужной строке, а на втором – через внутренний массив – к столбцу. Построчный просмотр зубчатого массива – не проблема, если сначала получить строку, а потом ее использовать. Однако просмотр по столбцам – неиссякаемый источник кэш-промахов.

Если число столбцов известно во время компиляции, можно легко совместить массив фиксированной длины с динамическим массивом:

```
enum size_t columns = 128;
// Определить матрицу с 64 строками и 128 столбцами
```

```
auto matrix = new double[columns][64];
// Не нужно выделять память под каждую строку - они и так уже существуют
foreach (ref row; matrix) {
    .. // Использовать строку типа double[columns]
}
```

В цикле из этого примера нужно обязательно использовать ключевое слово ref. Без него из-за передачи массива double[columns] по значению (см. раздел 4.2.4) создавалась бы копия каждой просматриваемой строки, что, скорее всего, отразилось бы на скорости выполнения кода.

Если во время компиляции известно число и строк, и столбцов многомерного массива, то можно использовать массив фиксированной длины массивов фиксированной длины, как в примере:

```
enum size_t rows = 64, columns = 128;
// Выделить память под матрицу с 64 строками и 128 столбцами
double[columns][rows] matrix;
// Вообще не нужно выделять память под массив - это значение
foreach (ref row; matrix) {
      // Использовать строку типа double[columns]
}
```

Чтобы получить доступ к элементу в строке i и столбце j, напишите matrix[i][j][1]. Немного странно, что в объявлении типа массива размеры измерений указаны «справа налево» (то есть double[столбцы][строки]), а при обращении к элементам массива индексы указываются «слева направо». Это объясняется тем, что [] и [n] в типах привязываются справа налево, а в выражениях – слева направо.

Сочетая массивы фиксированной длины с динамическими массивами, можно получать разнообразные многомерные массивы. Например, int[5][][15] – это трехмерный массив из 15 динамически размещаемых массивов, состоящих из блоков по 5 элементов типа int.

4.4. Ассоциативные массивы

Можно было бы представить массив как функцию, отображающую положительные целые числа (индексы) на значения некоторого произвольного типа (содержимое массива). Функция определена только для

[1] Заметим также, что переход по нужному индексу статического многомерного массива происходит за один раз, а сам массив хранится в непрерывной области памяти. Например, для хранения массива arr типа int[5][5] выделяется область размером 5 * 5 * int.sizeof байт, а переход по адресу arr[2][2] выглядит как &arr + 2 * 5 + 2. Если же статический массив размещается в сегменте данных (как глобальная переменная или как локальная с атрибутом static), а индексы известны на этапе компиляции, то переход по указателю вообще не потребуется. – *Прим. науч. ред.*

целых чисел на промежутке [0; длина_массива - 1] и задана в виде таблицы значений (собственно содержимого массива).

С этой точки зрения ассоциативные массивы – некая обобщенная форма массивов. В качестве области определения ассоциативных массивов можно использовать (почти) любой тип. Каждому значению из области определения можно поставить в соответствие значение другого типа – точно так же, как это делается с ячейками массивов. Метод записи, применяемый в случае ассоциативных массивов, и другие связанные с ними алгоритмы отличаются от метода и алгоритмов для других массивов, но, как и обычный массив, ассоциативный массив предоставляет возможность быстро сохранить и выбрать значение по ключу.

Как и ожидалось, тип ассоциативного массива задается как V[K], где K – тип ключей, а V – тип ассоциированных с ними значений. Например, создадим и инициализируем ассоциативный массив, который отображает строки на целые числа:

```
int[string] aa = [ "здравствуй":42, "мир":75 ];
```

Литерал ассоциативного массива (см. раздел 2.2.6) – это список разделенных запятыми пар вида ключ : значение, заключенный в квадратные скобки. В нашем примере литерал достаточно информативен, так что можно не указывать тип переменной aa явно, а просто написать:

```
auto aa = [ "здравствуй":42, "мир":75 ];
```

4.4.1. Длина

Для любого ассоциативного массива aa свойство aa.length типа size_t возвращает число ключей в aa (а значит, и число значений, учитывая, что между ключами и значениями отношение один-к-одному).

Ассоциативный массив, созданный по умолчанию (без литерала), имеет нулевую длину, а проверка на равенство такого массива константе null возвращает true.

```
string[int] aa;
assert(aa == null);
assert(aa.length == 0);
aa = [0:"zero", 1:"not zero"];
assert(aa.length == 2);
```

В отличие от одноименного свойства массивов, свойство .length ассоциативных массивов предназначено только для чтения. Тем не менее можно очистить ассоциативный массив, присвоив его переменной значение null.

4.4.2. Чтение и запись ячеек

Чтобы записать в ассоциативный массив aa новую пару ключ–значение, или заменить значение, уже поставленное в соответствие этому ключу, просто присвойте новое значение выражению aa[key][1], как здесь:

```
// Создать ассоциативный массив с соответствием строка/строка
auto aa = [ "здравствуй":"salve", "мир":"mundi" ];
// Перезаписать значения
aa["здравствуй"] = "ciao";
aa["мир"] = "mondo";
// Создать несколько новых пар ключ-значение
aa["капуста"] = "cavolo";
aa["моцарелла"] = "mozzarella";
```

Чтобы прочитать из ассоциативного массива значение по ключу, просто воспользуйтесь выражением aa[key]. (Компилятор различает чтение и запись и вызывает для этого функции, которые немного отличаются друг от друга.) Продолжим предыдущий пример:

```
assert(aa["здравствуй"] == "ciao");
```

Если вы попытаетесь прочитать значение по ключу, которого нет в ассоциативном массиве, возникнет исключительная ситуация. Но обычно генерация исключения в случае, когда ключ не обнаружен, – слишком строгая мера, чтобы быть полезной, поэтому для чтения ассоциативных массивов предоставляется альтернативная функция, возвращающая значение по умолчанию, если ключ не найден в массиве. Она реализована в виде метода get, принимающего два аргумента. Если при вызове aa.get(ключ, значение_по_умолчанию) в массиве найден ключ, то функция возвращает соответствующее ему значение, а выражение значение_по__умолчанию не вычисляется; иначе значение_по__умолчанию вычисляется и метод возвращает результат этого вычисления.

```
assert(aa["здравствуй"] == "ciao");
// Ключ "здравствуй" существует, поэтому второй аргумент игнорируется
assert(aa.get("здравствуй"  "salute") == "ciao");
// Ключ "здорово" не существует, возвратить второй аргумент
assert(aa.get("здорово"  "buongiorno") == "buongiorno");
```

Если вы просто хотите проверить, существует ли определенный ключ в ассоциативном массиве, воспользуйтесь оператором in[2]:

---

[1] При этом для массива типа V[K] передаваемые ключи должны иметь тип immutable(K) или неявно приводимый к нему. Это требование введено для того, чтобы в процессе работы программы значение ключа не могло быть изменено косвенным образом, что повлекло бы нарушение структуры ассоциативного массива. – *Прим. науч. ред.*

[2] Как уже говорилось, оператор in возвращает указатель на элемент, соответствующий ключу, или null, если такой ключ отсутствует в массиве. – *Прим. науч. ред.*

```
assert("здравствуй" in aa);
assert("эй" !in aa);
// Попытка прочесть aa["эй"] вызвала бы исключение
```

4.4.3. Копирование

Ассоциативный массив — это всего лишь ссылка с поверхностным копированием: при копировании или присваивании ассоциативных массивов создаются только новые псевдонимы для тех же данных внутри. Например:

```
auto a1 = [ "Jane":10.0, "Jack":20, "Bob":15 ];
auto a2 = a1;              // a1 и a2 ссылаются на одни данные
a1["Bob"] = 100;           // Изменяя a1,
assert(a2["Bob"] == 100); //   .мы изменяем a2.
a2["Sam"] = 3.5;           //   .и
assert(a1["Sam"] == 3.5); // наоборот
```

При этом у ассоциативных массивов, как и у обычных, есть свойство `.dup`, создающее поэлементную копию массива.

4.4.4. Проверка на равенство

Операторы `is`, `==` и `!=` работают так, как и можно было ожидать. Для двух ассоциативных массивов `a` и `b` одного и того же типа выражение `a is b` истинно тогда и только тогда, когда переменные `a` и `b` ссылаются на один и тот же ассоциативный массив (то есть одной из переменных было присвоено значение другой). Выражение `a == b` поочередно сравнивает пары ключ–значение двух массивов с помощью оператора `==`. Чтобы `a` и `b` были равны, необходимо, чтобы в них совпали все ключи и значения для этих ключей.

```
auto a1 = [ "Jane":10.0, "Jack":20, "Bob":15 ];
auto a2 = [ "Jane":10.0, "Jack":20, "Bob":15 ];
assert(a1 !is a2);
assert(a1 == a2);
a2["Bob"] = 18;
assert(a1 != a2);
```

4.4.5. Удаление элементов

Чтобы удалить из таблицы соответствий пару ключ–значение, передайте ключ в метод `remove`, имеющийся у каждого ассоциативного массива.

```
auto aa = [ "здравствуй":1, "до свидания":2 ];
aa.remove("здравствуй");
assert("здравствуй" !in aa);
aa.remove("эй"); // Ничего не происходит, т. к. в массиве aa нет ключа "эй"
```

Метод `remove` возвращает логическое значение: `true`, если удаленный ключ присутствовал в массиве, иначе — `false`.

4.4.6. Перебор элементов

Вы можете перебирать элементы ассоциативного массива с помощью старой доброй конструкции foreach (см. раздел 3.7.5). Пары ключ–значение просматриваются без определенного порядка:

```
import std.stdio;

void main() {
   auto coffeePrices = [
      "французская ваниль" : 262,
      "ява" : 239,
      "французская обжарка" : 224
   ];
   foreach (kind, price; coffeePrices) {
      writefln("%s стоит %s руб. за 100 г"  kind, price);
   }
}
```

Эта программа печатает стоимость разных сортов кофе:

```
французская ваниль стоит 262 руб. за 100 г
ява стоит 239 руб. за 100 г
французская обжарка стоит 224 руб. за 100 г
```

Свойство .keys массива позволяет скопировать сразу все ключи из этого массива. Для любого ассоциативного массива aa типа V[K] выражение aa.keys возвращает тип K[].

```
auto gammaFunc = [-1.5:2.363, -0.5:-3.545, 0.5:1.772];
double[] keys = gammaFunc.keys;
assert(keys == [ -1.5, 0.5, -0.5 ]);
```

Аналогично для любого ассоциативного массива aa свойство aa.values возвращает все значения из aa в виде массива типа V[]. В общем случае для перебора элементов ассоциативного массива предпочтительно использовать цикл foreach, а не свойства .keys и .values, так как обращение к любому из этих свойств требует выделения памяти под новый массив, причем довольно большого объема в случае больших ассоциативных массивов.

Есть два метода, позволяющих организовать перебор ключей и значений ассоциативного массива, не создавая новые массивы: с помощью выражения aa.byKey() можно просмотреть только ключи ассоциативного массива aa, а с помощью выражения aa.byValue() – только значения этого массива. Например:

```
auto gammaFunc = [-1.5:2.363, -0.5:-3.545, 0.5:1.772];
// Вывести все ключи
foreach (k; gammaFunc.byKey()) {
   writeln(k);
}
```

4.4.7. Пользовательские типы

Ассоциативные массивы организованы так, что для обеспечения быстрого поиска используют хеширование и сортировку ключей. Чтобы использовать пользовательский тип для ключей ассоциативного массива, для него необходимо определить два специальных метода: `toHash` и `opCmp`. Мы еще не научились определять собственные типы и методы, поэтому отложим этот разговор до главы 6.

4.5. Строки

К строкам в D особое отношение. Два решения, принятые на ранней стадии развития языка (еще при его определении), оказались выигрышными. Во-первых, в качестве своего стандартного набора знаков D принял Юникод. (А Юникод сегодня – самый популярный и всеобъемлющий стандарт определения и представления текстовых данных.) Во-вторых, D использует кодировки UTF-8, UTF-16 и UTF-32, не отдавая предпочтения ни одной из них и не препятствуя использованию в вашем коде любой другой кодировки.

Чтобы понять, как D работает с текстом, нужно кое-что знать о Юникоде и UTF. Если хотите изучить эти предметы в полном объеме, книга «Unicode Explained» [36] послужит вам полезным источником информации, а документированный стандарт «Консорциума Юникода» [56] – сейчас в пятом издании, что соответствует версии 5.1 стандарта Юникод, – самая полная и точная справка по стандарту.

4.5.1. Кодовые точки

Нужно пояснить: Юникод различает «абстрактный знак», или *кодовую точку* (*code point*), и его представление, или *кодировку* (*encoding*). Об этой тонкости мало кто знает, отчасти потому, что в стандарте ASCII нет такого отдельного представления. Старый добрый стандарт ASCII каждому знаку, часто встречающемуся в англоязычных текстах, (и каждому из немногих «управляющих кодов») ставит в соответствие число в диапазоне от 0 до 127, то есть 7 бит. Когда был предложен стандарт ASCII, большинство компьютеров уже использовали 8-битный байт (октет) в качестве адресуемой единицы, и вопрос о «кодировании» ASCII-текста не стоял. (Оставшийся бит оставлял простор для творчества, что закончилось «Кембрийским взрывом»[1] взаимно несовместимых расширений.)

[1] Кембрийский взрыв – неожиданное появление в раннекембрийских отложениях окаменелостей представителей многих подразделений животного царства на фоне отсутствия их окаменелостей или окаменелостей их предков в докембрийских отложениях. – *Прим. пер.*

Юникод же, напротив, сначала определяет кодовые точки, то есть, попросту говоря, числа, поставленные в соответствие абстрактным знакам. Абстрактный знак «А» получает номер 65, абстрактный знак € – номер 8364 и т. д. Принятие решений о том, какие знаки заслуживают быть включенными в таблицу знаков Юникода и как присваивать им номера, – одно из важных дел, которыми занимается организация «Консорциум Юникода». И это здорово, потому что все могут использовать установленное ею соответствие между абстрактными знаками и числами, не беспокоясь о таких мелочах, как его определение и документирование.

По версии стандарта Юникод 5.1 кодовые точки Юникод находятся в диапазоне от 0 до 1114111 (верхний предел гораздо чаще приводят в шестнадцатеричном представлении: 0x10FFFF, или U+10FFFF – в особой юникодовской форме записи). Возможно, обычное заблуждение о том, что двумя байтами можно представить любой из знаков таблицы Юникода, столь распространено из-за того, что некоторые языки приняли в качестве стандарта двухбайтное представление знаков (что, в свою очередь, стало следствием именно такого представления в более ранних версиях стандарта Юникод). На самом деле, число знаков Юникод ровно в 17 раз превышает 65536 (максимальное число, доступное для двухбайтного представления). (По правде говоря, кодовые точки с бóльшими значениями практически не используются, а многие из них вообще пока не имеют представления.)

В любом случае, когда дело касается кодовых точек, можно не думать об их представлении. Отвлеченно можно считать кодовые точки огромной таблицей значений функции, ставящей в соответствие целым числам от 0 до 1114111 абстрактные знаки. Порядок назначения номеров из этого диапазона имеет множество нюансов, но это не умаляет правильности нашего высокоуровневого описания. О конкретном представлении кодовой точки из таблицы Юникод в виде последовательности байтов позаботится *кодировка*.

4.5.2. Кодировки

Если бы Юникод не задумываясь последовал общему подходу ASCII, он бы просто расширил верхнюю границу 0x10FFFF до следующего байта, чтобы каждая кодовая точка представлялась бы тремя байтами. Но у такого решения есть определенный недостаток. В большинстве текстов на английском или другом языке с основанным на латинице алфавитом была бы задействована статистически очень малая часть от общего количества кодовых точек (чисел), то есть память тратилась бы понапрасну. Размер обычных текстов на латинице просто-напросто вырос бы втрое. Алфавиты с бóльшим количеством знаков (такие как азиатские системы письменности) нашли бы трем байтам лучшее применение, и это нормально, ведь в целом в тексте было бы меньше знаков (но каждый знак был бы более информативен).

Чтобы не занимать лишнее место, Юникод принял несколько схем кодирования *с переменной длиной* представления знаков. Такие схемы используют один или несколько «более узких» кодов для представления всего диапазона кодовых точек Юникода. Узкие коды (обычно 8- или 16-битные) называются *кодовыми единицами* (*code units*). Каждая кодовая точка представляется одной или несколькими кодовыми единицами.

Первой стандартизированной кодировкой, работающей по этому принципу, стала кодировка UTF-8. UTF-8, которую Кен Томпсон придумал однажды вечером в небольшом ресторанчике в Нью-Джерси [47], – почти образцовый пример оригинального и надежного решения. Основная идея UTF-8: использовать для кодирования любого заданного знака от 1 до 6 байт; добавлять управляющие биты, по которым можно будет различать представления знаков разной длины. Представления первых 127 кодовых точек в кодировке UTF-8 идентичны представлениям в ASCII. То есть все ASCII-тексты автоматически становятся корректными с точки зрения UTF-8, что само по себе блестящий ход. Для кодовых точек, не входящих в диапазон ASCII, UTF-8 использует представления разной длины (табл. 4.2).

Таблица 4.2. Битовые представления UTF-8. Длина представления определяется по контрольным битам, что позволяет выполнять синхронизацию посреди потока, восстановление после ошибок и просмотр строки в обратном направлении

Кодовая точка (в шестнадцатеричном представлении)	Бинарное представление
00000000–0000007F	0xxxxxxx
00000080–000007FF	110xxxxx 10xxxxxx
00000800–0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000–001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000–03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000–7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Поскольку на сегодня верхней границей диапазона кодовых точек Юникод является число 0x10FFFF, две последние последовательности зарезервированы для использования в будущем; в настоящее время корректны только четырехбайтные представления.

Выбранные последовательности управляющих битов обладают двумя любопытными свойствами:

1. Первый байт представления всегда отличается от остальных его байтов.
2. Первый байт однозначно определяет длину представления.

Первое свойство является ключевым, так как находит два важных применения. Первое – простая синхронизация: начав принимать передаваемую информацию в кодировке UTF-8, прямо посреди потока легко можно выяснить, где начинается представление следующей кодовой точки (просто найдите следующий байт, начинающийся не с 10). Второе применение этого свойства – просмотр в обратном направлении: по строке UTF-8 можно легко перемещаться от конца к началу, не сбиваясь. Возможность просматривать строки UTF-8 в обратном направлении позволяет организовать множество алгоритмов (например, эффективный поиск последнего вхождения одной строки в другую). Второе свойство последовательностей управляющих битов не столь значимо, но оно упрощает и ускоряет обработку строк.

В идеале часто встречающиеся кодовые точки должны иметь малые представления, а редко встречающиеся – большие. При таких условиях кодировка UTF-8 работает как хороший статистический кодировщик, обозначая более часто встречающиеся знаки меньшим количеством битов. Это удобно для языков с алфавитом на основе латиницы, где для большинства букв достаточно одного байта, а для редких букв с диакритическими знаками – двух.

UTF-16 – тоже кодировка с переменной длиной, но в ней применяется другой (пожалуй, менее элегантный) подход к кодированию. Кодовые точки со значениями в диапазоне от 0 до 0xFFFF кодируются одной 16-битной кодовой единицей, а кодовые точки со значениями в диапазоне от 0x10000 до 0x10FFFF представляются *суррогатными парами*, то есть двумя кодовыми единицами, первая из которых находится в диапазоне от 0xD800 до 0xDBFF, а вторая – в диапазоне от 0xDC00 до 0xDFFF. Ради этой кодировки Юникод отказался от отображения кодовых точек на значения в диапазоне 0xD800–0xDBFF. Диапазоны значений первой и второй кодовых единиц называются *верхней суррогатной зоной* (*high surrogate area*) и *нижней суррогатной зоной* (*low surrogate area*) соответственно.

Обычно UTF-16 критикуют за то, что в этой кодировке статистически редкие случаи также становятся наиболее сложными в обработке и требуют самого тщательного рассмотрения. К сожалению, не все, но большинство знаков Юникода – так называемая базовая многоязыковая плоскость (Basic Multilingual Plane, BMP) – действительно могут быть закодированы единственной кодовой единицей кодировки UTF-16, поэтому множество программ, работающих с UTF-16, автоматически принимают одну кодовую единицу за представление одного знака, отказываясь от проверок на наличие суррогатных пар в пользу эффективности. Еще больше усугубляет путаницу то, что некоторые языки изначально выбрали поддержку предшественницы UTF-16 – кодировки UCS-2 (в которой одной кодовой точке соответствуют ровно 16 бит), а позже решили добавить поддержку UTF-16, что осложнило использование старого кода, полагающегося на соответствие между знаками и их кодовыми единицами вида один-к-одному.

Наконец, кодировка UTF-32 использует 32 бита для одной кодовой единицы. Это означает, что в кодировке UTF-32 принято самое простое и легкое в использовании, но в то же время самое «прожорливое» представление кодовых точек. Обычно рекомендуют придерживаться следующей политики: кодировку UTF-8 использовать для хранения, а к кодировке UTF-32 обращаться лишь временно, во время обработки, и только при необходимости.

4.5.3. Знаковые типы

В языке D определено три знаковых типа: `char`, `wchar` и `dchar`, обозначающие кодовые единицы кодировок UTF-8, UTF-16 и UTF-32 соответственно. В качестве значений свойства `.init` этих типов намеренно выбраны некорректные значения: `char.init` равно `0xFF`, `wchar.init` — `0xFFFF`, а `dchar.init` — `0x0000FFFF`.

Из табл. 4.2 видно, что константа `0xFF` не может быть частью корректного битового представления знака в кодировке UTF-8, а значению `0xFFFF` Юникод намеренно не ставит в соответствие никакую кодовую точку.

Используемые по отдельности, значения этих трех знаковых типов ведут себя в основном как целые числа без знака и иногда могут использоваться для хранения некорректных UTF-представлений кодовых точек (компилятор не заботится о том, чтобы везде использовались корректные представления кодовых точек), но изначально задуманное назначение типов `char`, `wchar` и `dchar` — служить UTF-представлениями кодовых точек. А для работы с 8-, 16- и 32-битными целыми числами без знака и для представления кодировок, не входящих в группу UTF, лучше всего использовать типы `ubyte`, `ushort` и `uint` соответственно. Например, для работы с применявшимися до появления Юникода 8-битными кодовыми страницами вы можете взять за основу значения типа `ubyte`, а не `char`.

4.5.4. Массивы знаков + бонусы = строки

Сформированный массив любого знакового типа — такого как `char[]`, `wchar[]` или `dchar[]` — компилятор и библиотека средств поддержки времени исполнения считают строками Юникода в одной из UTF-кодировок. Следовательно, массивы знаков сочетают в себе мощь и гибкость, свойственные массивам, и некоторые дополнительные преимущества, предоставляемые Юникодом.

На самом деле, в D уже определены три типа строк, соответствующие трем размерам представления знаков: `string`, `wstring` и `dstring`. Это не особые типы, а всего лишь псевдонимы массивов знаковых типов с одним отличием: знаковый тип снабжен квалификатором `immutable`, запрещающим произвольное изменение отдельных знаков в строке. Например, тип `string` — более короткий синоним для типа `immutable(char)[]`. Подробное обсуждение квалификаторов типов (в том числе `immutable`)

мы отложим до главы 8, но для строк в любой кодировке действие immutable объясняется очень просто: свойства значения типа string, также известного как immutable(char)[], идентичны свойствам значения типа char[] (а свойства значения типа immutable(wchar)[] – свойствам значения типа wchar[]), за исключением маленького отличия: нельзя присвоить новое значение отдельному знаку строки:

```
string a = "hello";
char h = a[0]; // Все в порядке
a[0] = 'H';    // Ошибка!
               // Присваивать типу immutable(char) запрещено!
```

Чтобы изменить в строке какой-то конкретный знак, требуется создать новое значение типа string, применив конкатенацию:

```
string a = "hello";
a = 'H' ~ a[1    $]; // Все в порядке, делает выражение
                     // a == "Hello" истинным
```

Почему было принято такое решение? В конце концов в приведенном выше примере совершенно бессмысленно выделять новую область памяти под целую строку (вспомните, в разделе 4.1.6 говорилось, что оператор ~ всегда требует выделения новой области памяти под новый массив), вместо того чтобы просто изменить уже имеющуюся строку. В пользу квалификатора immutable говорит то, что его наличие упрощает ситуации, когда объект типа string, wstring или dstring копируется, а потом изменяется. Квалификатор гарантирует отсутствие лишних ссылок на одну и ту же строку. Например:

```
string a = "hello";
string b = a; // Переменная b теперь тоже указывает на значение "hello"
string c = b[0    4]; // Переменная c указывает на строку "hell"
// Если бы такое присваивание было разрешено, это изменило бы a, b, и c:
// a[0] = 'H';
// Конкатенация оставляет переменные b и c нетронутыми:
a = 'H' ~ a[1    $];
assert(a == "Hello" && b == "hello" && c == "hell");
```

Неизменяемость отдельных знаков позволяет работать с несколькими переменными, ссылающимися на одну и ту же строку, не боясь, что изменение одной из них отразится и на других. Копировать строковые объекты очень дешево, поскольку не реализуется никакая особая стратегия копирования (например, раннее копирование или копирование при записи).

Не менее весомая причина запретить изменения в строках на уровне кодовых единиц – такие изменения все равно лишены смысла. Элементы string имеют разную длину, а в большинстве случаев требуется заменить логические знаки (кодовые точки), а не физические (кодовые единицы), поэтому желание проводить хирургические операции над отдельными знаками возникает редко. Гораздо легче записать правильный UTF-код,

отказавшись от присваивания отдельным знакам, но уделив больше внимания работе с целыми строками и их фрагментами. Стандартная библиотека D задает тон, поддерживая работу со строками как с едиными сущностями (а не с индексами и отдельными знаками). Тем не менее писать UTF-код не так легко; например, в предыдущем примере в конкатенации 'H' ~ a[1 .. $] допущена ошибка: эта запись предполагает, что первая кодовая точка занимает ровно один байт. Правильное решение выглядит так:

```
a = 'H' ~ a[stride(a, 0)    $];
```

Функция stride из модуля std.utf стандартной библиотеки возвращает длину кода знака в указанной позиции строки. Для доступа к функции stride и другому полезному содержимому библиотеки вставьте где-нибудь ближе к началу программы строку:

```
import std.utf;
```

В нашем случае вызов stride(a, 0) возвращает количество байт двоичного представления первого знака (кодовой точки) в строке a. Именно это число мы используем при получении среза, помечая начало второго знака.

Наглядный пример поддержки Юникода языком можно обнаружить в строковых литералах, с которыми мы уже успели познакомиться (см. раздел 2.2.5). Строковые литералы D «понимают» кодовые точки из таблицы Юникод и автоматически кодируют их в соответствии с любой выбранной вами кодировкой. Например:

```
import std.stdio;

void main() {
   string a = "Независимо от представления \u03bb стоит \u20AC20.";
   wstring b = "Независимо от представления \u03bb стоит \u20AC20.";
   dstring c = "Независимо от представления \u03bb стоит \u20AC20.";
   writeln(a, '\n'  b, '\n'  c);
}
```

Несмотря на то что внутренние представления строк a, b и c сильно отличаются друг от друга, вам не нужно об этом беспокоиться, потому что вы задаете литерал в абстрактном виде, используя кодовые точки. Компилятор заботится обо всех тонкостях кодирования, так что в итоге программа печатает три строки с одним и тем же текстом:

```
Независимо от представления λ стоит €=20.
```

Кодировка литерала определяется контекстом, в котором этот литерал используется. В предыдущем примере компилятор преобразует строковый литерал, без какой-либо обработки во время исполнения программы, из кодировки UTF-8 в кодировку UTF-16, а потом в кодировку UTF-32 (соответствующие типам string, wstring и dstring), хотя написание литералов во всех трех случаях одинаково. Если требуемая кодиров-

ка литерала не может быть однозначно определена, добавьте к нему суффикс c, w или d (например, "как_здесь"d): строка будет преобразована в кодировку UTF-8, UTF-16 или UTF-32 соответственно (см. раздел 2.2.5.2).

4.5.4.1. Цикл foreach применительно к строкам

Если просматривать строку str (в любой кодировке) таким способом:

```
foreach (c; str) {
... // Использовать c
}
```

то переменная c поочередно примет значение каждой из *кодовых единиц* строки str. Например, если str – массив элементов типа char (с квалификатором immutable или без), то переменной c присваивается тип char. Это ожидаемо, если вспомнить, как ведет себя цикл просмотра с массивами, но иногда для строк такое поведение нежелательно. Например, напечатаем знаки строки типа string, заключив каждый из них в квадратные скобки.

```
void main() {
   string str = "Hall\u00E5, V\u00E4rld!";
   foreach (c; str) {
      write('[  c, ']');
   }
   writeln();
}
```

Но напечатает эта программа совсем не то, что ожидалось:

```
[H][a][l][l][?][?][,][ ][V][?][?][r][l][d][!]
```

Негатив знака ? (может отличаться в зависимости от операционной системы и используемого шрифта) – это немой протест консоли против отображения некорректного UTF-кода. Разумеется, попытка напечатать отдельный элемент типа char, обретающий смысл только в сочетании с другими элементами типа char, обречена на провал.

Но самое интересное начинается, если вы укажете для c другой знаковый тип. Например, назначим переменной c тип dchar:

```
  .тот же самый код, добавлен только тип "dchar"
foreach (dchar c; str) {
   write('[' c, ']');
}
```

В этом случае компилятор автоматически вставляет код для перекодировки «на лету» каждой кодовой единицы в str в представление, диктуемое типом переменной c. Наш цикл напечатает:

```
[H][a][l][l][å][,][ ][V][ä][r][l][d][!]
```

а это указывает на то, что каждый из двухбайтных знаков å и ä был правильно преобразован к соответствующему знаку типа dchar, и по-

этому они были напечатаны верно. То же самое будет напечатано, если задать для переменной c тип wchar, поскольку указанные в литерале два знака, отсутствующие в таблице ASCII, вмещаются в единственную кодовую единицу кодировки UTF-16, но это не общий случай (суррогатные пары будут обработаны неверно). Однако чтобы обеспечить максимально возможную степень безопасности, конечно же, лучше всего при просмотре строк использовать тип dchar.

В рассмотренном примере в инструкции foreach выполнялось перекодирование в направлении от «узкого» к более «широкому» представлению, но обратное преобразование также возможно. Например, можно начать со значения типа dstring, а затем просмотреть его по одному (закодированному) знаку типа char.

4.6. Опасный собрат массива – указатель

Объект массива отслеживает в памяти группу типизированных объектов, сохраняя адреса ее верхней и нижней границ. Указатель – «наполовину массив»: он позволяет отслеживать только один объект. Поэтому указатель не знает, где начинается и заканчивается группа объектов. Если вы получите эту информацию откуда-то извне, то сможете использовать ее для организации перемещения указателя, заставляя его указывать на соседние элементы.

Указатель на объект типа T обозначается как тип T* и по умолчанию имеет значение null (то есть указывает «в никуда»). Направить указатель на объект можно с помощью оператора получения адреса &, а использовать этот объект – с помощью оператора разыменования * (см. раздел 2.3.6.2). Например:

```
int x = 42;
int* p = &x;      // Получить адрес x
*p = 10;          // *p можно использовать там же, где и x
++*p;             // Обычные операторы также применимы
assert(x == 11); // Переменная x была изменена с помощью указателя p
```

Указатели могут участвовать в арифметических операциях, что делает чрезвычайно заманчивым их применение в качестве курсоров внутри массивов. Если увеличить указатель на единицу, он будет указывать на следующий элемент массива, если уменьшить на единицу – на предыдущий элемент. Прибавив к указателю целое число n, получим указатель на объект, отстоящий от элемента, на который указывал исходный указатель, на n позиций вправо, если n больше нуля, и влево, если n меньше нуля. Ради упрощения операции индексирования выражение p[n] эквивалентно выражению *(p + n). Наконец, разница между двумя указателями p2 - p1 соответствует такому целому числу n, что p1 + n == p2.

Можно получить адрес первого элемента массива arr с помощью выражения вида arr.ptr. Следовательно указатель на последний элемент

непустого массива arr можно получить с помощью выражения arr.ptr + arr.length - 1, а указатель на область памяти сразу за последним элементом массива – с помощью выражения arr.ptr + arr.length. Проиллюстрируем все сказанное примером:

```
auto arr = [ 5, 10, 20, 30 ];
auto p = arr.ptr;
assert(*p == 5);
++p;
assert(*p == 10);
++*p;
assert(*p == 11);
p += 2;
assert(*p == 30);
assert(p - arr.ptr == 3);
```

Однако будьте осторожны: если вы не обладаете информацией об адресах границ массива (указатель вам об этом не сообщит, а значит, это должно быть известно откуда-то еще), ситуация вскоре может стать непредсказуемой. Никакие операции с участием указателей не проверяются: указатель – это всего лишь адрес памяти длиной в слово[1], и арифметические операции, которые вы к нему применяете, просто слепо исполняют то, о чем вы просите. Это делает указатели невероятно быстрыми и при этом ужасно неосведомленными. Указатель недостаточно умен даже для того, чтобы понять, что он указывает на отдельный объект (в отличие от указания на элемент массива):

```
auto x = 10;
auto y = &x;
++y; // Хм...
```

Указателю также неизвестно, когда он вышел за границу массива:

```
auto x = [ 10, 20 ];
auto y = x.ptr;
y += 100;          // Хм.
*y = 0xdeadbeef; // Русская рулетка
```

Присваивать значение с помощью указателя, который не указывает на корректные данные, – значит играть в русскую рулетку с целостностью своей программы: записи могут «приземлиться» где угодно, растоптав самые тщательно оберегаемые данные, а то и код. Все это делает указатели *небезопасным для памяти* (*memory-unsafe*) средством.

Поэтому старательно избегайте указателей, отдавая предпочтение массивам, ссылкам на классы (см. главу 6), аргументам функций, переданным с ключевым словом ref (см. раздел 5.2.1), и автоматическому управ-

[1] В архитектуре *x*86 тип указатель размером в 4 байта соответствует двойному слову (DW), а слову соответствует тип short размером 2 байта. – *Прим. науч. ред.*

лению памятью. Все эти средства безопасны, могут эффективно проверяться и почти не снижают быстродействие.

В действительности, массивы – весьма полезная абстракция, тщательно спроектированная с единственною целью: *создать самое быстрое после указателей средство с учетом ограничений безопасности для памяти.* Очевидно, что сам по себе указатель не имеет доступа к достаточному количеству информации, чтобы выяснить что-то самостоятельно; массив, напротив, знает свой размер, поэтому может легко проверять, что все операции над ними совершаются в пределах границ расположения данных.

С точки зрения высокого уровня можно отметить, что массивы слишком низкоуровневые и что их реализация не дотягивает до абстрактного типа данных. С другой стороны, если мыслить низкоуровневыми категориями, может показаться, что в массивах нет необходимости, так как они могут быть реализованы с помощью указателей. Ответ на оба эти аргумента против массивов все тот же: «Я все объясню».

Ценность массивов в том, что из всех абстракций эта абстракция – самая низкоуровневая, но при этом она уже безопасна. Если бы язык предоставлял только указатели, то был бы не способен обеспечить безопасность различных пользовательских конструкций более высокого уровня, построенных на базе указателей. Массивы также не должны быть слишком высокоуровневыми, потому что являются встроенными, а значит, все остальное будет создаваться, используя их как основу. Хорошее встроенное средство должно быть низкоуровневым и быстрым, чтобы можно было строить на его базе абстракции более высокого уровня, необязательно столь же быстрые. Именно так и развиваются абстракции.

Существует «урезанная» безопасная версия D, известная как SafeD (см. главу 11), также есть флаг компилятора, установка которого включает проверку принадлежности используемых в программе инструкций и типов данных этому безопасному подмножеству средств языка. Естественно, в безопасном D (SafeD) запрещено большинство операций с указателями. Встроенные массивы – это важное средство, позволяющее создавать мощные, выразительные программы на SafeD.

4.7. Итоги и справочник

В табл. 4.3 собрана информация об операциях над динамическими массивами, в табл. 4.4. – об операциях над массивами фиксированной длины, а в табл. 4.5 – об операциях над ассоциативными массивами.

Таблица 4.3. Операции над динамическими массивами (a и b – два значения типа T[]; t, t_1, ..., t_k – значения типа T; n – значение, приводимое к типу размер_t)

Выражение	Тип	Описание
new T[n]	T[]	Создает массив (см. раздел 4.1)
[t_1,t_2, ..., t_k]	T[]	Литерал массива; T определяется по типу t_1 (см. разделы 2.2.6 и 4.1)
a = b	T[]	Присваивает один массив другому (см. раздел 4.1.4)
a[‹*в*›]	ref T	Предоставляет доступ к элементу по индексу (символ $ в выражении ‹*в*› заменяется на a.length, ‹*в*› должно быть приводимым к типу размер_t; кроме того, должно соблюдаться условие ‹*в*› < a.length) (см. раздел 4.1)
a[‹$в_1$› .. ‹$в_2$›]	T[]	Получает срез массива a (знак $ в ‹$в_1$› и ‹$в_2$› заменяется на a.length, ‹$в_1$› и ‹$в_2$› должны быть приводимыми к типу размер_t, также должно соблюдаться условие ‹$в_1$› <= ‹$в_2$› && ‹$в_2$› <= a.length) (см. раздел 4.1.3)
a[]	T[]	Поэлементная операция (см. раздел 4.1.7) или альтернативное написание выражения a[0 .. $], возвращающего содержимое всего массива
a.dup	T[]	Получает дубликат массива (см. раздел 4.1)
a.length	размер_t	Читает длину массива (см. раздел 4.1.10)
a.length = n	размер_t	Изменяет длину массива (см. раздел 4.1.1)
a is b	bool	Проверяет, идентичны ли массивы друг другу (см. разделы 4.1.5 и 2.3.4.3)
a !is b	bool	То же, что !(a is b)
a == b	bool	Поэлементно сравнивает массивы на равенство (см. раздел 4.1.5)
a != b	bool	То же, что !(a == b)
a ~ t	T[]	Конкатенирует массив и отдельное значение (см. раздел 4.1.6)
t ~ a	T[]	Конкатенирует отдельное значение и массив (см. раздел 4.1.6)
a ~ b	T[]	Конкатенирует два массива (см. раздел 4.1.6)
a ~= t	T[]	Присоединяет элемент к массиву (см. раздел 4.1.6)
a ~= b	T[]	Присоединяет один массив к другому (см. раздел 4.1.6)
a.ptr	T*	Возвращает адрес первого элемента массива a (небезопасная операция) (см. раздел 4.6)

Таблица 4.4. Операции над массивами фиксированной длины (a и b – два значения типа T[]; t, t_1, ..., t_k – значения типа T; n – значение, приводимое к типу размер_t)

Выражение	Тип	Описание
[t_1, ..., t_k]	T[k]	Литерал массива, но только если тип T[k] запрошен явно; T определяется по типу t_1 (см. разделы 2.2.6 и 4.1)
a = b	ref T[n]	Копирует содержимое одного массива в другой (см. раздел 4.2.4)
a[‹*в*›]	ref T	Предоставляет доступ к элементу по индексу (символ $ в ‹*в*› заменяется на a.length, ‹*в*› должно быть приводимым к типу размер_t; кроме того, должно соблюдаться условие ‹*в*› < a.length) (см. раздел 4.1)
a[‹$в_1$› .. ‹$в_2$›]	T[]/T[k]	Получает срез массива a (символ $ в ‹$в_1$› и ‹$в_2$› заменяется на a.length, ‹$в_1$› и ‹$в_2$› должны быть приводимыми к типу размер_t, также должно соблюдаться условие ‹$в_1$› <= ‹$в_2$› && ‹$в_2$› <= a.length) (см. раздел 4.2.3)
a[]	T[]	Поэлементная операция (см. раздел 4.1.7) или приведение a (массива фиксированной длины) к типу динамического массива, то же, что и a[0 .. $]
a.dup	T[]	Получает дубликат массива (см. раздел 4.2.4)
a.length	размер_t	Читает длину массива (см. раздел 4.2.1)
a is b	bool	Проверяет, идентичны ли массивы друг другу (см. разделы 4.2.5 и 2.3.4.3)
a !is b	bool	То же, что и !(a is b)
a == b	bool	Поэлементно сравнивает массивы на равенство (см. разделы 4.2.5 и 2.3.12)
a != b	bool	То же, что и !(a == b)
a ~ t	T[]	Конкатенирует массив и отдельное значение (см. раздел 4.2.6)
t ~ a	T[]	Конкатенирует отдельное значение и массив (см. раздел 4.2.6)
a ~ b	T[]	Конкатенирует два массива (см. раздел 4.2.6)
a.ptr	T*	Возвращает адрес первого элемента массива a (небезопасная операция)

Таблица 4.5. Операции над ассоциативными массивами (a и b – два значения типа V[K]; k, k_1, ..., k_1 – значения типа K; v, v_1, ..., v_k – значения типа V)

Операция	Тип	Описание
[t_1:v_1, ..., t_1:v_1]	V[K]	Литерал ассоциативного массива; K определяется по типу k_1, а V – по типу v_1 (см. разделы 2.2.6 и 4.4)
a = b	V[K]	Присваивает ассоциативный массив b переменной a типа «ассоциативный массив» (см. раздел 4.4.3)
a[k]	V	Предоставляет доступ к элементу по индексу (если ключ k не найден, возникает исключение) (см. раздел 4.4.2)
a[k] = v	V	Ставит в соответствие ключу k значение v (переопределяет предыдущее соответствие, если оно уже было назначено) (см. раздел 4.4.2)
k in a	V*	Ищет k в a, возвращает null, если не находит, иначе – указатель на значение, ассоциированное с k (см. раздел 4.4.2)
k !in a	bool	То же, что и !(k in a)
a.length	размер_t	Читает значение, соответствующее числу элементов в a (см. раздел 4.4.1)
a is b	bool	Проверяет, идентичны ли массивы друг другу (см. разделы 4.4.4 и 2.3.4.3)
a !is b	bool	То же, что и !(a is b)
a == b	bool	Поэлементно сравнивает массивы на равенство (см. разделы 4.4.4 и 2.3.12)
a != b	bool	То же, что и !(a == b)
a.remove(k)	bool	Удаляет пару с ключом k, если такая есть; возвращает true, если и только если ключ k присутствовал в a (см. раздел 4.4.5)
a.dup	V[K]	Создает дубликат ассоциативного массива a (см. раздел 4.4.3)
a.get(k, v)	V	Возвращает значение из a, соответствующее ключу k; по умолчанию возвращается значение v (см. раздел 4.4.2)
a.byKey()	int delegate(int delegate(ref K))	Возвращает делегат, пригодный для использования в цикле foreach для итерации по ключам
a.byValue()	int delegate(int delegate(ref V))	Возвращает делегат, пригодный для использования в цикле foreach для итерации по значениям

5

Данные и функции. Функциональный стиль

Обсуждать данные и функции сегодня, когда даже разговоры об объектах устарели, – это как вернуться в 1970-е. Но, к сожалению, все еще за горами день, когда говоришь компьютеру, что нужно сделать, и он сам выясняет, как это сделать. А пока этот день не настал, функции – обязательный компонент всех основных направлений программирования. По большому счету, любая программа состоит из вычислений, гоняющих данные туда-сюда; возводимые нами замысловатые строительные леса – типы, объекты, модули, фреймворки, шаблоны проектирования – только придают вычислениям нужные нам свойства, такие как модульность, изоляция ошибок или легкость сопровождения. Правильный язык программирования позволяет своему пользователю держаться золотой середины между кодом «для действия» и кодом «для существования». Идеальное соотношение зависит от множества факторов, из которых самый очевидный – размер программы: основная задача короткого скрипта – действовать, тогда как большое приложение вынуждено заниматься поддержкой неисполняемых вещей вроде интерфейсов, протоколов и модульных ограничений.

Благодаря своим мощным средствам моделирования D позволяет создавать объемистые программы; при этом он старается сократить до разумных пределов код «для существования», позволяя сосредоточиться на том, что нужно «для действия». Хорошо написанные функции на D, как правило, соединяют в себе компактность и универсальность, достигая порой ошеломляющей удельной мощности. Так что пристегнитесь, будем жечь резину.

5.1. Написание и модульное тестирование простой функции

Можно с полным основанием утверждать: главное, чем занимаются компьютеры (помимо скучных дел вроде ожидания ввода данных), – это *поиск*. Серверы и клиенты баз данных – ищут. Программы искусственного интеллекта – ищут. (А надоедливый болтун – банковский автоответчик? Тоже ищет.) Интернет-поисковики... ну, с этими ясно. Да и собственный опыт наверняка говорит вам, что, по сути, многие программы, якобы не имеющие с поиском ничего общего, на самом деле тоже довольно-таки много ищут. Какую бы задачу ни требовалось решить, всегда задействуется поиск. В свою очередь, многие оригинальные решения зависят от интеллектуальности и удобства программного поиска. Как и следовало ожидать, в мире вычислений полно понятий, имеющих отношение к поиску: сопоставление с шаблоном, реляционная алгебра, бинарный поиск, хеш-таблицы, бинарные деревья, префиксные деревья, красно-черные деревья, списки с пропусками... все это нам здесь никак не охватить, так что сейчас поставим цель поскромнее – определим несколько простых функций поиска на D, начав с простейшей из них – функции линейного поиска. Итак, без лишних слов напишем функцию, которая сообщает, содержит ли срез значений типа `int` определенное значение типа `int`.

```
bool find(int[] haystack, int needle) {
   foreach (v; haystack) {
      if (v == needle) return true;
   }
   return false;
}
```

Отлично. Поскольку это первое наше определение функции на D, опишем во всех подробностях, что именно она делает. Встретив определение функции `find`, компилятор приведет ее к более низкоуровневому представлению – скомпилирует в двоичный код. Во время исполнения программы при вызове функции `find` параметры `haystack` и `needle`[1] передаются в нее по значению. Это вовсе не означает, что если вы передадите в функцию массив из миллиона элементов, то он будет полностью скопирован; как отмечалось в главе 4, тип `int[]` (срез массива элементов типа `int`), который также называют *толстым указателем* (*fat pointer*), – это на самом деле пара «указатель + длина» или пара «указатель + указатель», которая хранит только границы указанного фрагмента массива. Из раздела 4.1.4 понятно, что передать в функцию `find` срез миллионного массива на самом деле означает передать в нее информацию, достаточную для получения адреса начала и конца этого среза. (Язык D и его стандартная библиотека широко поддерживают работу с контейнером

[1] Функция `find` ищет «иголку» (`needle`) в «стоге сена» (`haystack`). – *Прим. науч. ред.*

через его маленького, ограниченного представителя, который знает, как перемещаться по контейнеру. Обычно такой представитель называется *диапазоном*.) Так что в итоге в функцию `find` из вызывающего ее кода передаются только три машинных слова. Как только управление передано функции `find`, она делает свое дело и возвращает логическое значение (обычно в регистре процессора), которое вызвавший ее код уже готов получить. Что ж, как ободряюще говорят в конце телешоу «Большой ремонт», завершив какую-то неимоверно сложную работу: «Вот и все, что нужно сделать».

Если честно, в устройстве `find` есть кое-какие недостатки. Возвращаемое значение имеет тип `bool`, это очень неинформативно; также требуется информация о позиции найденного элемента, например, для продолжения поиска. Можно было бы возвращать целое число (и какое-нибудь особое значение, например `-1`, для случаев «элемент не найден»). Но хотя целые числа отлично подходят для доступа к элементам массива, занимающего непрерывную область памяти, они ужасно неэффективны с большинством других контейнеров (таких как связные списки). Чтобы добраться до `n`-го элемента связного списка после того, как функция `find` вернула `n`, понадобится пройти по списку элемент за элементом, начиная с его головы – то есть проделать почти ту же работу, что и сама операция поиска! Так что возвращать целое число в качестве результата – плохая идея в случае любой структуры данных, кроме массива.

Есть один способ, который сработает с разнообразными контейнерами – массивами, связными списками и даже с файлами и сокетами. Надо сделать так, чтобы функция `find` просто отщипывала по одному элементу («соломинке»?) от «стога сена» (`haystack`), пока не обнаружит искомое значение, и возвращала то, что останется от `haystack`. (Соответственно, если значение не найдено, функция `find` вернет опустошенный `haystack`.) Вот простая и обобщенная спецификация: «функция `find(haystack, needle)` сужает структуру данных `haystack` слева до тех пор, пока значение `needle` не станет началом, или до тех пор, пока не закончатся элементы в `haystack`, и затем возвращает остаток `haystack`». Давайте реализуем эту идею для типа `int[]`.

```
int[] find(int[] haystack, int needle) {
   while (haystack.length > 0 && haystack[0] != needle) {
      haystack = haystack[1    $];
   }
   return haystack;
}
```

Обратите внимание: функция `find` обращается только к первому элементу массива `haystack` и последовательно присваивает исходному массиву более узкое подмножество его самого. Эти примитивы потом легко можно заменить, скажем, специфичными для списков примитивами, но обобщением мы займемся чуть позже. А пока проверим, насколько хорошо работает полученная функция `find`.

В последние годы большинство методологий разработки уделяют все больше внимания правильному тестированию программного обеспечения. Это верное направление, поскольку тщательное тестирование действительно помогает отслеживать гораздо больше ошибок. В духе времени напишем короткий тест модуля, проверяющий работу нашей функции `find`. Просто вставьте следующий код после (как это сделал я) или до (как сделал бы фанат разработки через тестирование) определения функции `find`:

```
unittest {
   int[] a = [];
   assert(find(a, 5) == []);
   a = [ 1, 2, 3 ];
   assert(find(a, 0) == []);
   assert(find(a, 1).length == 3);
   assert(find(a, 2).length == 2);
   assert(a[0    $ - find(a, 3).length] == [ 1, 2 ]);
}
```

Все, что нужно сделать, чтобы получить работающий модуль, – это поместить функцию и тест модуля в файл `searching.d`, а затем ввести в командной строке:

```
$ rdmd --main -unittest searching.d
```

Если вы запустите компилятор с флагом `-unittest`, тесты модулей будут скомпилированы и подготовлены к запуску перед исполнением основной программы. Иначе компилятор проигнорирует все блоки `unittest`, что может быть полезно, если требуется запустить уже оттестированный код без задержек на начальном этапе. Флаг `--main` предписывает `rdmd` добавить ничего не делающую функцию `main`. (Если вы забыли написать `--main`, не волнуйтесь; компоновщик тут же витиевато напомнит вам об этом на своем родном языке – зашифрованном клингонском.) Заменитель функции `main` нужен нам, так как мы хотим запустить только тест модуля, а не саму программу. Ведь наш маленький файл может заинтересовать массу программистов, и они станут использовать его в своих проектах, в каждом из которых определена своя функция `main`.

5.2. Соглашения о передаче аргументов и классы памяти

Как уже говорилось, в функцию `find` передаются два аргумента (первый – типа `int`, а второй – толстый указатель, представляющий массив типа `int[]`), которые копируются в ее личные владения. Когда функция `find` возвращает результат, толстый указатель копируется обратно в вызывающий код. В этой последовательности действий легко распознать явный вызов по значению. В частности, изменения аргументов не будут «видны» инициатору вызова после того, как управление снова перейдет к нему. Однако остерегаться побочного эффекта все-таки нужно:

учитывая, что *содержимое* среза не копируется, изменения отдельных элементов среза *будут видны* инициатору вызова. Рассмотрим пример:

```
void fun(int x) { x += 42; }
void gun(int[] x) { x = [ 1, 2, 3 ]; }
void hun(int[] x) { x[0] = x[1]; }
unittest {
   int x = 10;
   fun(x);
   assert(x == 10);              // Ничего не изменилось
   int[] y = [ 10, 20, 30 ];
   gun(y);
   assert(y == [ 10, 20, 30 ]); // Ничего не изменилось
   hun(y);
   assert(y == [ 20, 20, 30 ]); // Изменилось!
}
```

Что же произошло? В первых двух случаях функции `fun` и `gun` изменили только собственные копии параметров. В частности, во втором случае толстый указатель был перенаправлен на другую область памяти, но исходный массив не был затронут. Однако в третьем случае функция `hun` решила изменить один элемент массива, и это изменение отразилось на исходном массиве. Это легко понять, представив, что срез `y` находится совсем не в том же месте, что и три целых числа, которыми `y` управляет. Так что если вы присвоите срез целиком, а-ля `x = [1, 2, 3]`, то срез, который раньше содержала переменная `x`, будет предоставлен самому себе, а `x` начнет новую жизнь; но если вы измените какой-то элемент `x[i]` среза `x`, то другие срезы, которым виден этот элемент (в нашем случае – в коде, вызвавшем `fun`), будут видеть и это изменение.

5.2.1. Параметры и возвращаемые значения, переданные по ссылке (с ключевым словом ref)

Иногда нам действительно нужно, чтобы изменения были видны в вызывающем коде. В этом случае поможет класс памяти `ref`:

```
void bump(ref int x) { ++x; }
unittest {
   int x = 1;
   bump(x);
   assert(x == 2);
}
```

Если функция ожидает значение по ссылке, то она принимает только «настоящие данные», а не временные значения. Все, что не является l-значением, отвергается во время компиляции. Например:

```
bump(5); // Ошибка! Нельзя передать r-значение по ссылке
```

Это предотвращает глупые ошибки – когда кажется, что дело сделано, а на самом деле вызов прошел безрезультатно.

Ключевым словом ref можно также снабдить результат функции. В этом случае за ним самим будет закреплен статус l-значения. Например, изменим функцию bump так:

```
ref int bump(ref int x) { return ++x; }
unittest {
   int x = 1;
   bump(bump(x)); // Два увеличения на 1
   assert(x == 3);
}
```

Внутренний вызов функции bump возвращает l-значение, поэтому такой результат можно правомерно использовать в качестве аргумента при внешнем вызове той же функции. Если бы определение bump выглядело так:

```
int bump(ref int x) { return ++x; }
```

то компилятор отверг бы вызов bump(bump(x)) как незаконную попытку привязать r-значение, возвращенное при вызове bump(x), параметру, передаваемому по ссылке при внешнем вызове bump.

5.2.2. Входные параметры (с ключевым словом in)

Параметр с ключевым словом in считается предназначенным только для чтения, его нельзя изменить никаким способом. Например:

```
void fun(in int x) {
   x = 42;        // Ошибка! Нельзя изменить параметр с ключевым словом in
}
```

Этот код не компилируется, то есть ключевое слово in накладывает на код достаточно строгие ограничения. Функция fun не может изменить даже собственную копию аргумента.

Практически неизменяемый параметр внутри функции, конечно, может быть полезен при анализе ее реализации, но еще более любопытный эффект наблюдается *за пределами* функции. Ключевое слово in запрещает даже косвенные изменения параметра, то есть те изменения, которые отражаются на объекте после того, как функция вернет управление вызвавшему ее коду. Это делает неизменяемые параметры невероятно полезными, поскольку они дают гарантии инициатору вызова, а не только внутренней реализации функции. Например:

```
void fun(in int[] data) {
   data = new int[10]; // Ошибка! Нельзя изменить неизменяемый параметр
   data[5] = 42;       // Ошибка! Нельзя изменить неизменяемый параметр
}
```

В первом случае ошибка неудивительна, поскольку она того же типа, что и приведенная выше ошибка с изменением отдельного значения типа int. Гораздо интереснее, почему возникла вторая ошибка. Неким магическим образом компилятор распространил действие ключевого

слова in с самого массива data на все его ячейки – то есть in обладает «глубоким» воздействием.

Ограничение, на самом деле, распространяется на любую глубину, а не только на один уровень. Проиллюстрируем сказанное примером с многомерным массивом:

```
// Массив массивов чисел имеет два уровня ссылок

void fun(in int[][] data) {
   data[5] = data[0];       // Ошибка! Нельзя изменить неизменяемый параметр
   data[5][0] = data[0][5]; // Ошибка! Нельзя изменить неизменяемый параметр
}
```

Так что ключевое слово in защищает свои данные от изменений *транзитивно*, полностью сверху донизу, учитывая все возможности косвенного доступа[1]. Такое поведение не является специфичным для массивов, оно распространяется на все типы данных языка D. В действительности, ключевое слово in в контексте параметра – это синоним квалификатора типа const[2], подробно описанного в главе 8.

5.2.3. Выходные параметры (с ключевым словом out)

Иногда параметры передаются по ссылке только для того, чтобы функция с их помощью что-то вернула. В таких случаях можно воспользоваться классом памяти out, напоминающим ref, – разница лишь в том, что перед входом в функцию out инициализирует свой аргумент значением по умолчанию (соответствующим типу аргумента):

```
// Вычисляет частное и остаток от деления для аргументов a и b.
// Возвращает частное по значению, а остаток - в параметре rem.
int divrem(int a, int b, out int rem) {
   assert(b != 0);
   rem = a % b;
   return a / b;
}
unittest {
   int r;
   int d = divrem(5, 2, r);
   assert(d == 2 && r == 1);
}
```

---

[1] Следует подчеркнуть, что проверка выполнения подобных соглашений выполняется на этапе компиляции, и если компилятор обмануть, например с помощью приведения типов, то соглашения можно нарушить. Пример: (cast(int[])data)[5] = 42; даст именно то, что ожидается. Но это уже моветон. – *Прим. науч. ред.*

[2] На самом деле, in означает scope const, однако семантика scope не до конца продумана и, возможно, в дальнейшем scope вообще исчезнет из языка. – *Прим. науч. ред.*

В этом коде можно было бы с тем же успехом вместо ключевого слова out использовать ref, поскольку выбор out всего лишь извещает инициатора вызова, что функция divrem не ожидает от параметра rem осмысленного значения.

5.2.4. Ленивые аргументы (с ключевым словом lazy)[1]

Порой значение одного из аргументов функции требуется лишь в исключительном случае, а в остальных вычислять его не нужно и хотелось бы избежать напрасных усилий. Рассмотрим пример:

```
bool verbose; // Флаг, контролирующий отладочное журналирование
void log(string message)
{
   // Если журналирование включено, выводим строку на экран
if (verbose)
      writeln(message);
}

int result = foo(); log("foo() returned " ~ to!string(result));
```

Как видим, вычислять выражение "foo() returned " ~ to!string(result) нужно, только если переменная verbose имеет значение true. При этом выражение, передаваемое этой функции в качестве аргумента, будет вычислено в любом случае. В данном примере это конкатенация двух строк, которая потребует выделения памяти и копирования в нее содержимого каждой из них. И все это для того, чтобы узнать, что переменная verbose имеет значение false и значение аргумента никому не нужно! Можно было бы передавать вместо строки делегат, возвращающий строку (делегаты описаны в разделе 5.6.1):

```
void log(string delegate() message)
{
   if (verbose)
      writeln(message());
}
  .log({return "foo() returned " ~ to!string(result);});
```

В этом случае аргумент будет вычислен, только если он действительно нужен, но такая форма слишком громоздка. Поэтому D вводит такое понятие, как «ленивые» аргументы. Такие аргументы объявляются с атрибутом lazy, выглядят как обычные аргументы, но вычисляются только тогда, когда требуется их значение.

```
void log(lazy string message)
{
```

[1] Описание этой части языка намеренно не было включено в оригинал книги, но поскольку эта возможность есть в текущих реализациях языка, мы добавили ее описание. – *Прим. науч. ред.*

```
    if (verbose)
        writeln(message); // Значение message вычисляется здесь
}
```

5.2.5. Статические данные (с ключевым словом static)

Несмотря на то что ключевое слово static не имеет отношения к передаче аргументов функциям, разговор о нем здесь к месту, поскольку, как и ref, атрибут static данных определяет *класс памяти*, то есть несколько подробностей хранения этих данных.

Любое объявление переменной может быть дополнено ключевым словом static. В этом случае *для каждого потока исполнения* будет создана собственная копия этой переменной. Рациональное обоснование и последствия этого отступления от установленной языком C традиции выделять единственную копию static-переменной для всего приложения обсуждаются в главе 13.

Статические данные сохраняют свое значение между вызовами функций независимо от места их определения (внутри или вне функции). Выбор размещения статических данных в разнообразных контекстах касается только видимости, но не хранения. На уровне модуля данные с атрибутом static в действительности обрабатываются так же, как и данные с атрибутом private.

```
static int zeros; // Практически то же самое, что и private int zeros;

void fun(int x) {
    static int calls;
    ++calls;
    if (!x) ++zeros;

}
```

Статические данные должны быть инициализированы константами[1], вычисляемыми во время компиляции. Инициализировать статические данные уровня функции при первом ее вызове можно с помощью простого трюка, который использует в качестве напарника статическую логическую переменную:

```
void fun(double x) {
    static double minInput;
    static bool minInputInitialized;
    if (!minInputInitialized) {
        minInput = x;
        minInputInitialized = true;
    } else {
```

[1] На самом деле, их *можно* инициализировать только константами, а можно вообще не инициализировать (тогда они принимают значение по умолчанию). – *Прим. науч. ред.*

```
        if (x < minInput) minInput = x;
    }

}
```

5.3. Параметры типов

Вернемся к функции find, определенной в разделе 5.1, поскольку в ней есть немало спорных моментов. Во-первых, эта функция может быть полезна только в довольно редких случаях, поэтому стоит поискать возможность ее обобщения. Начнем с простого наблюдения. Присутствие в find типа int – это пример жесткого кодирования, простого и ясного. В логике кода ничего не изменится, если придется искать значения типа double в срезах типа double[] или значения типа string в срезах типа string[]. Поэтому можно попробовать заменить тип int некой заглушкой – параметром функции find, который описывал бы тип, а не значение задействованных сущностей. Чтобы воплотить эту идею, нужно привести наше определение к следующему виду:

```
T[] find(T)(T[] haystack, T needle) {
    while (haystack.length > 0 && haystack[0] != needle) {
        haystack = haystack[1    $];
    }
    return haystack;
}
```

Как и ожидалось, тело функции find не претерпело никаких изменений, изменилась только сигнатура. Теперь в ней две пары круглых скобок: в первой перечислены параметры типов функции, а вторая содержит обычный список параметров, которые могут воспользоваться только что определенными параметрами типов. Теперь можно обрабатывать не только срезы элементов типа int, но срезы *чего угодно* (неважно, встроенные это или пользовательские типы). В довершение наш предыдущий тест unittest продолжает работать, так как компилятор автоматически выводит тип T из типов аргументов. Чисто сработано! Но не станем почивать на лаврах и добавим тест модуля, который бы подтверждал оправданность этих повышенных ожиданий:

```
unittest {
    // Проверка способностей к обобщению
    double[] d = [ 1.5, 2.4 ];
    assert(find(d, 1.0) == null);
    assert(find(d, 1.5) == d);
    string[] s = [ "one"  "two" ];
    assert(find(s, "two") == [ "two" ]);
}
```

Что же происходит, когда компилятор видит усовершенствованное определение функции find? Компилятор сталкивается с гораздо более сложной задачей, чем в случае с аргументом типа int[], потому что теперь T

еще неизвестен – это может быть какой угодно тип. А разные типы записываются по-разному, передаются по-разному и щеголяют разными определениями оператора ==. Решить эту задачу очень важно, поскольку параметры типов действительно открывают новые перспективы и в разы расширяют возможности для повторного использования кода. В настоящее время наиболее распространены два подхода к генерации кода для параметризации типов [43]:

- *Гомогенная трансляция*: все данные приводятся к общему формату, что позволяет скомпилировать единственную версию `find`, которая подойдет всем.
- *Гетерогенная трансляция*: при каждом вызове `find` с различными аргументами типов (`int`, `double`, `string` и т. д.) компилятор генерирует отдельную версию `find` для каждого использованного типа.

Гомогенная трансляция подразумевает, что язык обязан предоставить универсальный интерфейс доступа к данным, которым воспользуется `find`. А гетерогенная трансляция больше напоминает помощника, пишущего по одному варианту функции `find` для каждого формата данных, который вам может встретиться, при этом все варианты он строит по одной заготовке. Очевидно, что у обоих этих подходов есть как преимущества, так и недостатки, о чем нередко ведутся жаркие споры в разных программистских сообществах. Плюсы гомогенной трансляции – универсальность, простота и компактность сгенерированного кода. Например, в чисто функциональных языках все представляется в виде списков, а во многих чисто объектно-ориентированных языках – в виде объектов; в обоих случаях предлагается универсальный доступ к данным. Тем не менее гомогенной трансляции свойственны такие недостатки, как строгость, недостаток выразительности и неэффективность. Гетерогенная трансляция, напротив, отличается специализированностью, выразительной мощью и скоростью сгенерированного кода. Плата за это – распухание готового кода, усложнение языка и неуклюжая модель компиляции (обычный упрек в адрес гетерогенных подходов – что они представляют собой «возвеличенный макрос» [вздох]; а поскольку благодаря C макрос считается чем-то нехорошим, этот ярлык придает гетерогенной компиляции сильный негативный оттенок).

Тут стоит обратить внимание на одну деталь: гетерогенная трансляция включает гомогенную по той простой причине, что «один формат» входит в «множество форматов», а «одна реализация» – в «множество реализаций». На этом основании (все прочие спорные моменты пока отложим) можно утверждать, что гетерогенная трансляция мощнее гомогенной. При наличии средства гетерогенной трансляции ничто не мешает, по крайней мере теоретически, использовать один универсальный формат данных и одну универсальную функцию, когда захочется. Обратное, при использовании гомогенного подхода, просто невозможно. Тем не менее наивно было бы считать гетерогенные подходы «лучшими», поскольку кроме выразительной мощи есть другие аргументы, которые также нельзя упускать из виду.

D использует гетерогенную трансляцию (внимание, ожидается бомбардировка техническими терминами) с поиском статически определенных идентификаторов и отложенной проверкой типов. Это означает, что, встретив определение обобщенной функции `find`, компилятор D выполняет синтаксический разбор ее тела, сохраняет результаты, запоминает место определения функции – и больше ничего, до тех пор пока кто-нибудь не вызовет `find`. В этот момент компилятор извлекает разобранное определение `find` и пытается скомпилировать его, подставив тип, который инициатор вызова передал взамен `T`. Если функция использует идентификаторы (символы), компилятор ищет их в том контексте, где была определена эта функция.

Если компилятор не смог сгенерировать функцию `find` для этого конкретного типа, генерируется сообщение об ошибке. Что на самом деле довольно неприятно, поскольку исключение может возникнуть из-за незамеченной ошибки в `find`. Зато теперь у нас есть веский повод прочесть следующий раздел, потому что `find` содержит две ошибки – не функциональные, а связанные с обобщенностью: теперь понятно, что функция `find` одновременно и излишне, и недостаточно обобщенна. Посмотрим, как работает этот дзэнский тезис.

5.4. Ограничения сигнатуры

Допустим, у нас есть массив с элементами типа `double`, в котором мы хотим найти целое число. Казалось бы, все должно пройти довольно гладко:

```
double[] a = [ 1.0, 2.5, 2.0, 3.4 ];
a = find(a, 2); // Ошибка! Не определена функция find(double[], int)
```

Вот мы и в западне. В данной ситуации функция `find` ожидает значение типа `T[]` в качестве первого аргумента и значение типа `T` в качестве второго. Тем не менее `find` получает значение типа `double[]` и значение типа `int`, то есть `T = double` и `T = int` соответственно. Если мы достаточно пристально вглядимся в этот код, то, конечно же, заметим, что инициатор вызова в действительности хотел использовать в качестве `T` тип `double` и собирался реализовать свою задумку, рассчитывая на аккуратное неявное приведение значения типа `int` к типу `double`. Тем не менее заставлять язык пытаться комбинаторно выполнить сразу и неявное преобразование, и вывод типов – в общем случае рискованное предприятие, поэтому D все это проделать не пытается. Раз вы сказали `T[]` и `T`, то не можете передать `double[]` и `int`.

Похоже, нашей реализации функции `find` недостает обобщенности, поскольку она требует, чтобы типы среза и искомого значения были идентичны. А на самом деле для заданного типа среза мы должны принимать *любое* значение, сравнимое с элементом среза с помощью оператора `==`.

Один параметр типа — хорошо, а два параметра типа — лучше:

```
T[] find(T, E)(T[] haystack, E needle) {
   while (haystack.length > 0 && haystack[0] != needle) {
      haystack = haystack[1    $];
   }
   return haystack;
}
```

Теперь функция проходит тест на ура. Но технически полученная функция `find` лжет, поскольку заявляет, что принимает абсолютно любые `T` и `E`, в том числе их бессмысленные сочетания! Чтобы показать, почему эту неточность нужно считать проблемой, рассмотрим следующий вызов:

```
assert(find([1, 2, 3], "Hello")); // Ошибка!
   // Сравнение haystack[0] != needle некорректно для int[] и string
```

Компилятор действительно обнаруживает проблему; однако находит он ее в сравнении, расположенном в теле функции `find`. Это может смутить неосведомленного пользователя, поскольку неясно, где именно возникает ошибка: в месте вызова функции `find` или в ее теле. (В частности, имя файла и номер строки, возвращенные в отчете компилятора, прямо указывают внутрь определения функции `find`.) Если источник проблемы находится в конце длинной цепочки вызовов, ситуация становится еще более запутанной. Хотелось бы это исправить. Итак, в чем же корень всех бед? В переносном смысле, функция `find` выписывает чеки, которые ее тело не может обналичить.

В своей сигнатуре (это часть кода до первой фигурной скобки {) функция `find` торжественно заявляет, что принимает срез любого типа `T` и значение любого типа `E`. Компилятор радостно с этим соглашается, отправляет в `find` бессмысленные аргументы, устанавливает типы (`T = int` и `E = string`) и на этом успокаивается. Но как только дело доходит до тела `find`, компилятор смущенно обнаруживает, что не может сгенерировать осмысленный код для сравнения `haystack[0] != needle`, и выводит сообщение об ошибке примерно следующего содержания: «Функция `find` откусила больше, чем может прожевать». Тело `find` в действительности может принять только некоторые из всех возможных сочетаний типов `T` и `E` — те, которые можно проверять на равенство.

Можно было бы реализовать какой-то страховочный механизм. Но D выбрал другое решение: разрешить автору `find` систематически ограничивать применимость функции. Верное место для указания ограничения такого рода — сигнатура функции `find`, как раз там, где `T` и `E` появляются впервые. Для этого в D применяется *ограничение сигнатуры* (*signature constraint*):

```
T[] find(T, E)(T[] haystack, E needle)
   if (is(typeof(haystack[0] != needle) == bool))
{
```

```
        // Реализация остается той же
}
```

Выражение `if` в сигнатуре во всеуслышание заявляет, что функция `find` примет параметр `haystack` типа `T[]` и параметр `needle` типа `E`, только если выражение `haystack[0] != needle` возвращает логический тип. У этого ограничения есть ряд важных последствий. Во-первых, выражение `if` проясняет для автора, компилятора и читателя, чего именно функция `find` ждет от своих параметров, избавляя всех троих от необходимости исследовать тело функции (обычно куда более объемное, чем у нашей). Во-вторых, с выражением `if` в качестве буксира функция `find` теперь легко отклонит вызов при попытке передать параметры, не поддающиеся сравнению, что, в свою очередь, позволяет гладко срабатывать другим средствам языка, таким как перегрузка функций. В-третьих, новое определение помогает компилятору конкретизировать свои сообщения об ошибках: теперь очевидно, что ошибка происходит при обращении к функции `find`, а не в ее теле.

Заметим, что выражение, к которому применяется оператор `typeof`, никогда не вычисляется во время исполнения программы; оператор лишь определяет тип выражения, если оно скомпилируется. (Если выражение с оператором `typeof` не компилируется, то это не ошибка компиляции, а просто сигнал, что рассматриваемое выражение не имеет никакого типа, а «никакого типа» – это не `bool`.) В частности, не стоит беспокоиться о том, что в проверку вовлечено значение `haystack[0]`, даже если длина `haystack` равна нулю. И обратно: в ограничении сигнатуры запрещается использовать условия, не вычислимые во время компиляции программы; например, нельзя ограничить функцию `find` условием `needle > 0`.

5.5. Перегрузка

Мы определили функцию `find`, чтобы определить срез и элемент. А теперь напишем новую версию функции `find`, которая сообщает, можно ли найти один срез в другом. Обычный подход к решению этой проблемы – поиск полным перебором, с двумя вложенными циклами. Такой алгоритм не очень эффективен: время его работы пропорционально произведению длин рассматриваемых срезов. Но мы пока не будем беспокоиться об эффективности алгоритма, а сосредоточимся на определении хорошей сигнатуры для только что добавленной функции. Предыдущий раздел снабдил нас практически всем, что нужно. И действительно, сама собой напрашивается реализация:

```
T1[] find(T1, T2)(T1[] longer, T2[] shorter)
   if (is(typeof(longer[0    1] == shorter) : bool))
{
   while (longer.length >= shorter.length) {
      if (longer[0    shorter.length] == shorter) break;
      longer = longer[1    $];
   }
```

```
    return longer;
}
```

Ага! Как видите, на этот раз мы не попали в западню – не сделали функцию слишком специализированной. Не самое лучшее определение выглядело бы так:

```
// Нет! Эта сигнатура слишком строгая!
bool find(T)(T[] longer, T[] shorter) {

}
```

Оно, конечно, немного короче, но зато на порядок строже. Наша реализация, не копируя данные, может сказать, содержит ли срез элементов типа `int` срез элементов типа `long`, а срез элементов типа `double` – срез элементов типа `float`. Упрощенной сигнатуре эти возможности были просто недоступны. Вам бы пришлось или повсюду копировать данные, чтобы гарантировать наличие на месте нужных типов, или вообще отказаться от затеи с общей функцией и выполнять поиск вручную. А что это за функция, если она хорошо смотрится в игрушечных примерах и не справляется с серьезной нагрузкой!

Поскольку мы добрались до реализации, заметим уже хорошо знакомое сужение среза `longer` по одному элементу слева (во внешнем цикле). Задача внутреннего цикла – сравнение массивов `longer[0 .. shorter.length] == shorter`, где сравниваются первые `shorter.length` элементов среза `longer` с элементами среза `shorter`.

D поддерживает перегрузку функций: несколько функций могут разделять одно и то же имя, если отличаются числом аргументов или типом хотя бы одного из них. Во время компиляции правила языка определяют, какая именно функция должна быть вызвана. Перегрузка основана на нашей врожденной лингвистической способности избавляться от двусмысленности в значении слов, используя контекст. Это средство языка позволяет предоставить обширную функциональность, избегая соответствующего роста количества терминов, которые должен запомнить инициатор вызовов. С другой стороны, если правила выбора реализации функции при вызове слишком неопределенны, люди могут думать, что вызывают одну функцию, а на самом деле будут вызывать другую. А если упомянутые правила, наоборот, сделать слишком жесткими, программисту придется искажать логику своего кода, объясняя компилятору, какую функцию он имел в виду. D старается сохранить простоту правил, и в этом конкретном случае применяемое правило не является заумным: если вычисление ограничения сигнатуры функции (выражения `if`) возвращает `false`, функция просто удаляется из множества перегрузки – ее вообще перестают рассматривать как претендента на вызов. Для наших двух версий функции `find` соответствующие выражения `if` никогда не являются истинными одновременно (с одними и теми же аргументами). Так что при любом вызове `find` по крайней мере один вариант перегрузки себя скрывает; никогда не возникает двусмысленность,

над которой нужно ломать голову. Итак, продолжим ход своей мысли с помощью теста модуля:

```
unittest {
   // Проверим, как работает новая версия функции find
   double[] d1 = [ 6.0, 1.5, 2.25, 3 ];
   float[] d2 = [ 1.5, 2.25 ];
   assert(find(d1, d2) == d1[1    $]);
}
```

Неважно, где расположены эти две функции `find`: в одном или разных файлах; между ними никогда не возникнет соревнование, поскольку выражения `if` в ограничениях их сигнатур никогда не являются истинными одновременно. Продолжая обсуждение правил перегрузки, представим, что мы очень много работаем с типом `int[]` и хотим определить для него оптимизированный вариант функции `find`:

```
int[] find(int[] longer, int[] shorter) {
   ...
}
```

В этой записи версия функции `find` не имеет параметров типа. Кроме того, вполне ясно, что между обобщенной версией `find`, которую мы определили выше, и специализированной версией для целых значений происходит некое состязание. Каково относительное положение этих двух функций в пищевой цепи перегрузки и какой из них удастся захватить вызов ниже?

```
int[] ints1 = [ 1, 2, 3, 5, 2 ];
int[] ints2 = [ 3, 5 ];
auto test = find(ints1, ints2); // Корректно или ошибка?
                                // Обобщенная или специализированная?
```

Подход D к решению этого вопроса очень прост: выбор всегда падает на более специализированную функцию. Однако в более общем случае понятие «более специализированная» требует некоторого объяснения; оно подразумевает, что существует некоторое отношение порядка специализированности, «меньше или равно» для функций. И оно существует на самом деле; это отношение называется *отношением частичного порядка на множестве функций* (*partial ordering of functions*).

5.5.1. Отношение частичного порядка на множестве функций

Судя по названию, без черного пояса по матан-фу с этим не разобраться, а между тем отношение частичного порядка — очень простое понятие. Считайте это распространением знакомого нам числового отношения $\leq$ на другие множества, в нашем случае на множество функций. Допустим, есть две функции foo_1 и foo_2, и нужно узнать, является ли foo_1 чуть менее подходящей для вызова, чем foo_2 (вместо «foo_1 подходит меньше, чем foo_2» будем писать $foo_1 \leq foo_2$). Если определить такое отношение, то

у нас появится критерий, по которому можно определить, какая из функций выигрывает в состязании за вызов при перегрузке: при вызове foo можно будет отсортировать всех претендентов с помощью отношения ≤ и выбрать самую «большую» из найденных функцию foo. Чтобы частичный порядок работал в полную силу, это отношение должно быть рефлексивным (a ≤ a), антисимметричным (если a ≤ b и b ≤ a, считается, что a и b идентичны) и транзитивным (если a ≤ b и b ≤ c, то a ≤ c).

D определяет отношение частичного порядка на множестве функций очень просто: если функция foo_1 может быть вызвана с типами параметров foo_2, то $foo_1 \leq foo_2$. Возможны случаи, когда $foo_1 \leq foo_2$ и $foo_2 \leq foo_1$ одновременно; в таких ситуациях говорится, что функции *одинаково специализированны*. Например:

```
// Три одинаково специализированных функции: любая из них
// может быть вызвана с типом параметра другой
void sqrt(real);
void sqrt(double);
void sqrt(float)
```

Эти функции одинаково специализированны, поскольку любая из них может быть вызвана как с типом float, так и с double или real (как ни странно, это разумно, несмотря на неявное преобразование с потерями, см. раздел 2.3.2).

Также возможно, что ни одна из функций не ≤ другой; в этом случае говорится, что foo_1 и foo_2 *неупорядочены*.[1] Можно привести множество случаев неупорядоченности, например:

```
// Две неупорядоченные функции: ни одна из них
// не может быть вызвана с типом параметра другой.
void print(double);
void print(string);
```

Нас больше всего интересуют случаи, когда истинно ровно одно неравенство из пары $foo_1 \leq foo_2$ и $foo_2 \leq foo_1$. Пусть истинно первое неравенство, тогда говорится, что функция foo_1 менее специализированна, чем функция foo_2. А именно:

```
// Две упорядоченные функции: write(double) менее специализированна,
// чем write(int), поскольку первая может быть вызвана с int,
// а последняя не может быть вызвана с double.
void write(double);
void write(int);
```

Ввод отношения частичного порядка позволяет D принимать решение относительно перегруженного вызова $foo(arg_1, \ldots, arg_n)$ по следующему простому алгоритму:

[1] Именно этот момент делает «частичный порядок» «частичным». В случае отношения полного порядка (например ≤ для действительных чисел) неупорядоченных элементов нет.

1. Если существует всего одно соответствие (типы и количество параметров соответствуют списку аргументов), то использовать его.
2. Сформировать множество кандидатов $\{foo_1, \ldots, foo_k\}$, которые бы принимали вызов, если бы другие перегруженные версии вообще не существовали. Именно на этом шаге срабатывает механизм определения типов и вычисляются условия в ограничениях сигнатур.
3. Если полученное множество пусто, то выдать ошибку «нет соответствия».
4. Если не все функции из сформированного множества определены в одном и том же модуле, то выдать ошибку «попытка кроссмодульной перегрузки».
5. Исключить из множества претендентов на вызов все функции, менее специализированные по сравнению с другими функциями из этого множества; оставить только самые специализированные функции.
6. Если оставшееся множество содержит больше одной функции, выдать ошибку «двусмысленный вызов».
7. Единственный элемент множества – победитель.

Вот и все. Рассмотрим первый пример:

```
void transmogrify(uint) {}
void transmogrify(long) {}

unittest {
   transmogrify(42); // Вызывает transmogrify(uint)
}
```

Здесь нет точного соответствия, можно применить любую из функций, поэтому на сцене появляется частичное упорядочивание. Из него следует, что, несмотря на способность обеих функций принять вызов, первая из них более специализированна, поэтому победа присуждается ей. (Хорошо это или плохо, но `int` автоматически приводится к `uint`.) А теперь добавим в наш набор обобщенную функцию:

```
// То же, что и выше, плюс
void transmogrify(T)(T value) {}

unittest {
   transmogrify(42);       // Как и раньше, вызывает transmogrify(uint)
   transmogrify("hello"); // Вызывает transmogrify(T), T=string
   transmogrify(1.1);      // Вызывает transmogrify(T), T=double
}
```

Что же происходит, когда функция `transmogrify(uint)` сравнивается с функцией `transmogrify(T)(T)` на предмет специализированности? Хотя было решено, что `T = int`, во время сравнения `T` не заменяется на `int`, обобщенность сохраняется. Может ли функция `transmogrify(uint)` принять некоторый произвольный тип `T`? Нет, не может. Поэтому можно сделать вывод, что версия `transmogrify(T)(T)` менее специализированна,

чем transmogrify(uint), так что обобщенная функция исключается из множества претендентов на вызов. Итак, в общем случае предпочтение отдается необобщенным функциям, даже когда для их применения требуется неявное приведение типов.

5.5.2. Кроссмодульная перегрузка

Четвертый шаг алгоритма из предыдущего раздела заслуживает особого внимания. Вот немного измененный пример с перегруженными версиями для типов uint и long (разница лишь в том, что задействовано больше файлов):

```
// В модуле calvin.d
void transmogrify(long) { ... }
// В модуле hobbes.d
void transmogrify(uint) { ... }

// Модуль client.d
import calvin, hobbes;
unittest {
   transmogrify(42);
}
```

Перегруженная версия transmogrify(uint) из модуля hobbes.d является более специализированной; но компилятор все же отказывается вызвать ее, диагностируя двусмысленность. D твердо отвергает кроссмодульную перегрузку. Если бы такая перегрузка была разрешена, то значение вызова зависело бы от взаимодействия множества включенных модулей (в общем случае может быть много модулей, много перегруженных версий и больше сложных вызовов, за которые будет вестись борьба). Представьте: вы добавляете в работающий код всего одну новую команду import – и его поведение изменяется непредсказуемым образом! Кроме того, если разрешить кроссмодульную перегрузку, читать код явно станет на порядок труднее: чтобы выяснить, какая функция будет вызвана, нужно будет знать, что содержит не один модуль, а все включенные модули, поскольку в каком-то из них может быть определено лучшее соответствие. И даже хуже: если бы имел значение порядок определений на верхнем уровне, вызов вида transmogrify(5) мог бы в действительности завершиться вызовом различных функций в зависимости от их расположения в файле. Кроссмодульная перегрузка – это неиссякаемый источник проблем, поскольку подразумевает, что при чтении фрагмента кода нужно постоянно держать в голове большой меняющийся контекст.

Один модуль может содержать группу перегруженных версий, реализующих нужную функциональность для разных типов. Второй модуль может вторгнуться, только чтобы добавить что-то новое к этой функциональности. Однако второй модуль может определять собственную группу перегруженных версий. Пока функция в одном модуле не начинает угонять вызовы, которые по праву должны были принадлежать

функциям другого модуля, двусмысленность не возникает. До вызова функции нет возможности узнать, существует ли конфликт. Рассмотрим пример:

```
// В модуле calvin.d
void transmogrify(long) { .. }
void transmogrify(uint) {    }

// В модуле hobbes.d
void transmogrify(double) {    }

// В модуле susie.d
void transmogrify(int[]) {    }
void transmogrify(string) {    }

// Модуль client.d
import calvin, hobbes, susie;

unittest {
   transmogrify(5);          // Ошибка! кроссмодульная перегрузка,
                             // затрагивающая модули calvin и hobbes.
   calvin.transmogrify(5); // Все в порядке, точное требование,
                             // вызвана calvin.transmogrify(uint)
   transmogrify(5.5);        // Все в порядке, только hobbes
                             // может принять этот вызов.
   transmogrify("привет"); // Привет от Сьюзи
}
```

Кельвин, Хоббс и Сьюзи взаимодействуют интересными способами. Обратите внимание, насколько тонки различия между двусмысленностями в примере; первый вызов порождает конфликт между модулями calvin.d и hobbes.d, но это совершенно не значит, что эти модули взаимно несовместимы: третий вызов проходит гладко, поскольку ни одна функция в других модулях не в состоянии обслужить его. Наконец, модуль susie.d определяет собственные перегруженные версии и никогда не конфликтует с остальными двумя модулями (в отличие от одноименных персонажей комикса[1]).

Управление перегрузкой

Где бы вы ни встретили двусмысленность из-за кроссмодульной перегрузки, вы всегда можете указать направление перегрузки одним из двух основных способов. Первый – уточнить свою мысль, снабдив имя функции именем модуля, как это показано на примере второго вызова calvin.transmogrify(5). Поступив так, вы ограничите область поиска функции единственным модулем calvin.d. Внутри этого модуля также действуют правила перегрузки. Более очевидный способ – назначить проблемному идентификатору *локальный псевдоним*. Например:

[1] Речь о ежедневном комиксе американского художника Билла Уоттерсона «Кельвин и Хоббс». – *Прим. пер.*

```
// Внутри calvin.d
import hobbes;
alias hobbes.transmogrify transmogrify;
```

Эта директива делает нечто весьма интересное: она свозит все перегруженные версии transmogrify из модуля hobbes.d в модуль calvin.d. Так что если модуль calvin.d содержит упомянутую директиву, то можно считать, что, помимо собственных перегруженных версий, он определяет все перегруженные версии, которые определял hobbes.d. Это очень мило со стороны модуля calvin.d: он демократично советуется с модулем hobbes.d всякий раз, когда нужно принять решение, какая версия transmogrify должна быть вызвана. Иначе, если бы модулям calvin.d и hobbes.d не повезло и они решили бы игнорировать существование друг друга, модуль client.d все равно мог бы вызвать transmogrify, назначив псевдонимы обеим перегруженным версиям (и calvin.transmogrify, и hobbes.transmogrify).

```
// Внутри client.d
alias calvin.transmogrify transmogrify;
alias hobbes.transmogrify transmogrify;
```

Теперь при любом вызове transmogrify из модуля client.d решение о перегрузке будет приниматься так, будто перегруженные версии transmogrify, определенные в модулях calvin.d и hobbes.d, присутствуют в модуле client.d.

5.6. Функции высокого порядка. Функциональные литералы

Мы уже знаем, как найти элемент или срез в другом срезе. Однако под поиском не всегда подразумевается просто поиск заданного значения. Задача может быть сформулирована и так: «Найти в массиве чисел первый отрицательный элемент». Несмотря на все свое могущество, наша библиотека поиска не в состоянии выполнить это задание.

Основная идея функции find в том, что она ищет значение, удовлетворяющее некоторому логическому условию, или предикату; до сих пор в роли предиката всегда выступало сравнение на равенство (оператор ==). Однако более гибкая функция find может принимать предикат от пользователя и выстраивать логику линейного поиска вокруг него. Если удастся наделить функцию find такой мощью, она превратится в *функцию высокого порядка*, то есть функцию, которая может принимать другие функции в качестве аргументов. Это очень мощный подход к решению задач, поскольку объединяя собственную функциональность и функциональность, предоставляемую ее аргументами, функция высокого порядка достигает гибкости поведения, не доступной простым функциям. Чтобы заставить функцию find принимать предикат, воспользуемся *параметром-псевдонимом*.

```
T[] find(alias pred, T)(T[] input)
   if (is(typeof(pred(input[0])) == bool))
{
   for (; input.length > 0; input = input[1    $]) {
      if (pred(input[0])) break;
   }
   return input;
}
```

Эта новая перегруженная версия функции find принимает не только «классический» параметр, но и загадочный параметр-псевдоним alias pred. Параметру-псевдониму можно поставить в соответствие любой аргумент: значение, тип, имя функции – все, что можно выразить знаками. А теперь посмотрим, как вызывать эту новую перегруженную версию функции find.

```
unittest {
   int[] a = [ 1, 2, 3, 4, -5, 3, -4 ];
   // Найти первое отрицательное число
   auto b = find!(function bool(int x) { return x < 0; })(a);
}
```

На этот раз функция find принимает два списка аргументов. Первый список отличается синтаксисом !(...) и содержит обобщенные аргументы. Второй список содержит классические аргументы. Обратите внимание: несмотря на то что функция find объявляет два обобщенных параметра (alias pred и T), вызывающий ее код указывает только один аргумент. Вызов имеет такой вид, поскольку никто не отменял работу механизма определения типов: по контексту автоматически определяется, что T = int. До этого момента при наших вызовах find никогда не возникало необходимости указывать какие-либо обобщенные аргументы: компилятор определял их за нас. Однако на этот раз автоматически определить pred невозможно, поэтому мы указали его в виде функционального литерала. Функциональный литерал – это запись

```
function bool(int x) { return x < 0; }
```

где function – ключевое слово, а все остальное – обычное определение функции, только без имени.

Функциональные литералы (также известные как анонимные функции, или лямбда-функции) очень полезны во множестве ситуаций, однако их синтаксис сложноват. Длина литерала в наше примере – 41 знак, но только около 5 знаков занимаются настоящим делом. Чтобы решить эту проблему, D позволяет серьезно урезать синтаксис. Первое сокращение – это уничтожение возвращаемого типа и типов параметров: компилятор достаточно умен, чтобы определить их все, поскольку тело анонимной функции всегда под рукой.

```
auto b = find!(function(x) { return x < 0; })(a);
```

Второе сокращение – изъятие собственно ключевого слова function. Можно применять оба сокращения одновременно, как это сделано здесь (получается очень сжатая форма записи):

```
auto b = find!((x) { return x < 0; })(a);
```

Эта запись абсолютно понятна для посвященных, в круг которых вы вошли пару секунд назад.

5.6.1. Функциональные литералы против литералов делегатов

Важное требование к механизму лямбда-функций: он должен разрешать доступ к контексту, в котором была определена лямбда-функция. Рассмотрим слегка измененный вариант:

```
unittest {
   int[] a = [ 1, 2, 3, 4, -5, 3, -4 ];
   int z = -2;
   // Найти первое число меньше z
   auto b = find!((x) { return x < z; })(a);
   assert(b == a[4    $]);
}
```

Этот видоизмененный пример работает, что уже о многом говорит. Но если, просто ради эксперимента, вставить перед функциональным литералом ключевое слово, код загадочным образом перестает работать!

```
auto b = find!(function(x) { return x < z; })(a);
// Ошибка! Функция не может получить доступ к кадру стека вызывающей функции!
```

Что же происходит и что это за жалоба о кадре стека? Очевидно, должен быть какой-то внутренний механизм, с помощью которого функциональный литерал получает доступ к переменной z – он не может чудом добыть ее расположение из воздуха. Этот механизм закодирован в виде скрытого параметра – *указателя на кадр стека*, принимаемого литералом. Компилятор использует указатель на кадр стека, чтобы осуществлять доступ к внешним переменным, таким как z. Тем не менее функциональному литералу, который *не* использует никаких локальных переменных, не требуется дополнительный параметр. Будучи статически типизированным языком, D должен различать эти случаи, и он действительно различает их. Кроме функциональных литералов есть еще литералы делегатов, которые создаются так:

```
unittest {
   int z = 3;
   auto b = find!(delegate(x) { return x < z; })(a); // OK
}
```

В отличие от функций, делегаты имеют доступ к включающему их фрейму. Если в литерале нет ключевых слов function и delegate, компилятор автоматически определяет, какое из них подразумевалось. И снова на

помощь приходит механизм определения типов по контексту, позволяя самому сжатому, самому удобному коду еще и автоматически делать то, что нужно.

```
auto f = (int i) {};
assert(is(f == function));
```

5.7. Вложенные функции

Теперь можно вызывать функцию find с произвольным функциональным литералом, что довольно изящно. Но если литерал сильно разрастается или появляется желание использовать его несколько раз, становится неудобно писать тело функции в месте ее вызова (предположительно несколько раз). Хотелось бы вызывать find с именованной функцией (а не анонимной); кроме того, желательно сохранить право доступа к локальным переменным на случай, если понадобится к ним обратиться. Для этой и многих других задач D предоставляет такое средство, как вложенные функции.

Определение вложенной функции выглядит точно так же, как определение обычной функции, за исключением того, что вложенная функция объявляется внутри другой функции. Например:

```
void transmogrify(int[] input, int z) {
   // Вложенная функция
   bool isTransmogrifiable(int x) {
      if (x == 42) {
         throw new Exception("42 нельзя трансмогрифировать");
      }
      return x < z;
   }
   // Найти первый изменяемый элемент в массиве input
   input = find!(isTransmogrifiable)(input);
   ...
   // ...и снова
   input = find!(isTransmogrifiable)(input);
    ..
}
```

Вложенные функции могут быть очень полезны во многих ситуациях. Не делая ничего свыше того, что может сделать обычная функции, вложенная функция повышает удобство и модульность, поскольку расположена прямо внутри функции, которая ее использует, и имеет доступ к ее контексту. Последнее преимущество особенно важно; если бы в рассмотренном примере нельзя было воспользоваться вложенностью, получить доступ к z было бы гораздо сложнее.

Применив тот же трюк, что и функциональный литерал (скрытый параметр), вложенная функция isTransmogrifiable получает доступ к фрейму стека своего родителя, в частности к переменной z. Иногда может понадобиться заведомо избежать таких обращений к родительскому фрейму,

превратив isTransmogrifiable в самую обычную функцию, за исключением места ее определения (внутри transmogrify). Для этого просто добавьте перед определением isTransmogrifiable ключевое слово static (а какое еще?):

```
void transmogrify(int[] input, int z) {
   static int w = 42;
   // Вложенная обычная функция
   static bool isTransmogrifiable(int x) {
      if (x == 42) {
         throw new Exception("42 нельзя трансмогрифировать ");
      }
      return x < w; // Попытка обратиться к z вызвала бы ошибку
   }
    ..
}
```

Теперь, с ключевым словом static в качестве буксира, функции isTransmogrifiable доступны лишь данные, определенные на уровне модуля, и данные внутри transmogrify, также помеченные ключевым словом static (как показано на примере переменной w). Любые данные, которые могут изменяться от вызова к вызову, такие как параметры функций или нестатические переменные, недоступны (но, разумеется, могут быть переданы явно).

5.8. Замыкания

Как уже говорилось, alias – это чисто символическое средство; все, что оно делает, – придает одному идентификатору значение другого. В нашем предыдущем примере pred – это не настоящее значение, так же как и имя функции – это не значение; pred нельзя ничего присвоить. Если требуется создать массив функций (например, последовательность команд), ключевое слово alias не поможет. Здесь определенно нужно что-то еще, и это не что иное, как возможность иметь осязаемый объект функции, который можно записывать и считывать, сильно напоминающий указатель на функцию в C.

Рассмотрим, например, такую непростую задачу: «Получив значение x типа T, возвратить функцию, которая находит первое значение, равное x, в массиве элементов типа T». Подобное химически чистое, косвенное определение типично для функций высокого порядка: вы ничего *не делаете* сами, а только возвращаете то, что должно быть сделано. То есть нужно написать функцию, которая (внимание) возвращает другую функцию, которая, в свою очередь, принимает параметр типа T[] и возвращает значение типа T[]. Итак, возвращаемый тип функции, которую мы собираемся написать, – T[] delegate(T[]). Почему delegate, а не function? Как отмечалось выше, вдобавок к своим аргументам делегат получает доступ еще и к состоянию, в котором он определен, а функция – только

к аргументам. А наша функция как раз должна обладать некоторым состоянием, поскольку необходимо как-то сохранять значение x.

Это очень важный момент, поэтому его следует подчеркнуть. Представьте, что тип T[] function(T[]) – это просто адрес функции (одно машинное слово). Эта функция обладает доступом только к своим параметрам и глобальным переменным программы. Если передать двум указателям на одну и ту же функцию одни и те же аргументы, они получат доступ к одному и тому же состоянию программы. Любой, кто пробовал работать с обратными вызовами (callbacks) C – например, для оконных систем или запуска потоков, – знаком с вечной проблемой: указатели на функции не имеют доступа к собственному локальному состоянию. Способ, который обычно применяется в C для того, чтобы обойти эту проблему, – использование параметра типа void* (нетипизированный адрес), через который и передается информация о состоянии. Другие системы обратных вызовов, вроде старой капризной библиотеки MFC, сохраняют дополнительное состояние в глобальном ассоциативном массиве, третьи, такие как Active Template Library (ATL), динамически создают новые функции с помощью ассемблера. Везде, где необходимо взаимодействовать с обратными вызовами C, применяются некоторые решения, позволяющие обратным вызовам получать доступ к локальным состояниям; это далеко не простая задача.

С ключевым словом delegate все эти проблемы испаряются. Делегаты достигают этого ценой своего размера: делегат хранит указатель на функцию и указатель на окружение этой функции. Хотя это и больше по весу и порой медленнее, но в то же время и значительно мощнее. Так что в собственных разработках гораздо предпочтительнее использовать делегаты, а не функции. (Конечно же, функция вида function незаменима при взаимодействии с C через обратные вызовы.)

Теперь, когда уже так много сказано, попробуем написать новую функцию – finder. Не забудем, что вернуть нужно T[] delegate(T[]).

```
import std.algorithm;

T[] delegate(T[]) finder(T)(T x)
   if (is(typeof(x == x) == bool))
{
   return delegate(T[] a) { return find(a, x); };
}
unittest {
   auto d = finder(5);
   assert(d([1, 3, 5, 7, 9]) == [ 5, 7, 9 ]);
   d = finder(10);
   assert(d([1, 3, 5, 7, 9]) == []);
}
```

Трудно не согласиться, что такие вещи, как две команды return в одной строке, для непосвященных всегда будут выглядеть странновато. Что ж,

при первом знакомстве причудливой наверняка покажется не только эта функция высокого порядка. Так что начнем разбирать функцию finder построчно: она параметризирована с помощью типа T, принимает обычный параметр типа T и возвращает значение типа T[] delegate(T[]); кроме того, на T налагается ограничение: два значения типа T должны быть сравнимы, а результат сравнения должен быть логическим. (Как и раньше, «глупое» сравнение x == x здесь только ради типов, а не для каких-то определенных значений.) Затем finder разумно делает свое дело, возвращая литерал делегата. У этого литерала короткое тело, в котором вызывается наша ранее определенная функция find, завершающая выполнение условий поставленной задачи. Возвращенный делегат называется *замыканием* (*closure*).

Порядок использования функции finder ожидаем: ее вызов возвращает делегат, который потом можно вызвать и которому можно присваивать новые значения. Переменная d, определенная в тесте модуля, имеет тип T[] delegate(T[]), но благодаря ключевому слову auto этот тип можно не указывать явно. На самом деле, если быть абсолютно честным, с помощью ключевого слова auto можно сократить и определение finder; все типы присутствовали в нем лишь для облегчения понимания примера. Вот гораздо более краткое определение функции finder:

```
auto finder(T)(T x) if (is(typeof(x == x) == bool)) {
   return (T[] a) { return find(a, x); };
}
```

Обратите внимание на использование ключевого слова auto вместо возвращаемого типа функции, а также на то, что ключевое слово delegate опущено; компилятор с радостью позаботится обо всем этом за нас. Тем не менее в литерале делегата запись T[] указать необходимо. Ведь компилятор должен за что-то зацепиться, чтобы сотворить волшебство, обещанное ключевым словом auto: возвращаемый тип делегата определяется по типу функции find(a, x), который, в свою очередь, определяется по типам a и x; в результате такой цепочки выводов делегат приобретает тип T[] delegate(T[]), этот же тип возвращает функция finder. Без знания типа a вся эта цепочка рассуждений не может быть осуществима.

5.8.1. Так, это работает... Стоп, не должно... Нет, все же работает!

Наш тест модуля unittest помогает исследовать поведение функции finder, но, конечно же, не доказывает корректность ее работы. Важный и совсем неочевидный вопрос: возвращаемый функцией finder делегат использует значение x, а где находится x после того, как finder вернет управление? На самом деле, в этом вопросе слышится серьезное опасение за происходящее (ведь D использует для вызова функций обычный стек вызовов): инициатор вызова вызывает функцию finder, x отправляется на вершину стека вызовов, функция finder возвращает результат, стек восстанавливает свое состояние до вызова finder... а значит, возвра-

щенный функцией finder делегат использует для доступа адрес в стеке, по которому уже нет нужного значения!

«Продолжительность жизни» локального окружения (в нашем случае окружение состоит только из x, но оно может быть сколь угодно большим) – это классическая проблема реализации замыканий, и каждый язык, поддерживающий замыкания, должен ее как-то решать. В языке D применяется следующий подход[1]. В общем случае все вызовы используют обычный стек. А обнаружив замыкание, компилятор автоматически копирует используемый контекст в кучу и устанавливает связь между делегатом и областью памяти в куче, позволяя ему использовать расположенные в ней данные. Выделенная в куче память подлежит сбору мусора.

Недостаток такого подхода в том, что каждый вызов finder порождает новое требование выделить память. Тем не менее замыкания очень выразительны и позволяют применить многие интересные парадигмы программирования, поэтому в большинстве случаев затраты более чем оправданны.

5.9. Не только массивы. Диапазоны. Псевдочлены

Раздел 5.3 закончился загадочным утверждением: «функция find одновременно и излишне, и недостаточно обобщенна». Затем мы узнали, почему функция find излишне обобщенна, и исправили эту ошибку, наложив дополнительные ограничения на типы ее параметров. Пришло время выяснить, почему эта функция все же недостаточно обобщенна.

В чем смысл линейного поиска? В поисках заданного значения или значения, удовлетворяющего заданному условию, просматриваются элементы указанной структуры данных. Проблема в том, что до сих пор мы работали только с непрерывными массивами (срезами, встречающимися в нашем определении find в виде T[]), но к понятию линейного поиска непрерывность не имеет никакого отношения. (Она имеет отношение только к механизмам организации просмотра.) Ограничившись типом T[], мы лишили функцию find доступа ко множеству других структур данных, с которыми может работать алгоритм линейного поиска. Язык, предлагающий, к примеру, сделать find методом некоторого типа Array («массив»), вполне заслуживает вашего скептического взгляда. Это не значит, что решить задачу с помощью этого языка невозможно; просто наверняка поработать пришлось бы гораздо больше, чем это необходимо.

Пора начать все с нуля, пересмотрев нашу базовую реализацию find. Для удобства приведем ее здесь:

```
T[] find(T)(T[] haystack, T needle) {
   while (haystack.length > 0 && haystack[0] != needle) {
```

[1] Тот же подход используют ML и другие реализации функциональных языков.

```
        haystack = haystack[1    $];
    }
    return haystack;
}
```

Какие основные операции мы применяем к массиву haystack **и что означает каждая из них?**

1. haystack.length > 0 **сообщает, остались ли еще элементы в** haystack.
2. haystack[0] **осуществляет доступ к первому элементу** haystack.
3. haystack = haystack[1 .. $] **исключает из рассмотрения первый элемент** haystack.

Конкретный способ, каким массивы реализуют эти операции, непросто распространить на другие контейнеры. Например, проверять с помощью выражения haystack.length > 0, **есть ли в односвязном списке элементы, – подход, достойный премии Дарвина[1]. Если не обеспечено постоянное кэширование длины списка (что по многим причинам весьма проблематично), то для вычисления длины списка таким способом потребуется время, пропорциональное самой длине списка, а быстрое обращение к началу списка занимает всего лишь несколько машинных инструкций. Применить к спискам индексацию – столь же проигрышная идея. Так что выделим сущность рассмотренных операций, представим полученный результат в виде трех именованных функций и оставим их реализацию типу** haystack. **Примерный синтаксис базовых операций, необходимых для реализации алгоритма линейного поиска:**

1. haystack.empty **– для проверки** haystack **на пустоту.**
2. haystack.front **– для получения первого элемента** haystack.
3. haystack.popFront() **– для исключения из рассмотрения первого элемента** haystack.

Обратите внимание: первые две операции не изменяют haystack **и потому не используют круглые скобки, третья же операция изменяет** haystack, **и синтаксически это отражено в виде скобок** (). **Переопределим функцию** find, **применив в ее определении новый блестящий синтаксис:**

```
R find(R, T)(R haystack, T needle)
    if (is(typeof(haystack.front != needle) == bool))
{
    while (!haystack.empty && haystack.front != needle) {
        haystack.popFront();
    }
    return haystack;
}
```

[1] **Премия Дарвина – виртуальная премия, ежегодно присуждаемая тем, кто наиболее глупым способом лишился жизни или способности к зачатию, в результате не внеся свой вклад в генофонд человечества (и тем самым улучшив его).** – *Прим. пер.*

Было бы неплохо сейчас погреться в лучах этого благотворного определения, если бы не суровая реальность: тесты модулей не проходят. Да и могло ли быть иначе, когда встроенный тип среза `T[]` и понятия не имеет о том, что нас внезапно осенило и мы решили определить новое множество базовых операций с произвольными именами `empty`, `front` и `popFront`. Мы должны определить их для всех типов `T[]`. Естественно, все они будут иметь простейшую реализацию, но они все равно нам нужны, чтобы заставить нашу милую абстракцию снова заработать с тем типом данных, с которого мы начали.

5.9.1. Псевдочлены и атрибут @property

Наша синтаксическая проблема заключается в том, что все вызовы функций до сих пор выглядели как `функция(аргумент)`, а теперь мы хотим определить такие вызовы: `аргумент.функция()` и `аргумент.функция`, то есть *вызов метода* и *обращение к свойству* соответственно. Как мы узнаем из следующего раздела, для пользовательских типов они определяются довольно-таки просто, но `T[]` – это встроенный тип. Как же быть?

Язык D видит в этом чисто синтаксическую проблему и разрешает ее посредством нотации псевдочленов: если компилятор встретит запись `a.функция(b, c, d)`, где `функция` не является членом типа значения `a`, он заменит этот вызов на `функция(a, b, c, d)`[1] и попытается обработать вызов в этой новой форме. (При этом попытки обратного преобразования не предпринимаются: если вы напишете `функция(a, b, c, d)` и это окажется бессмыслицей, версия `a.функция(b, c, d)` не проверяется.) Предназначение псевдометодов – позволить вызывать обычные функции с помощью знакомого кому-то из нас синтаксиса «отправить-сообщение-объекту». Итак, без лишних слов реализуем `empty`, `front` и `popFront` для встроенных массивов. Для этого хватит трех строк:

```
@property bool empty(T)(T[] a) { return a.length == 0; }
@property ref T front(T)(T[] a) { return a[0]; }
void popFront(T)(ref T[] a) { a = a[1    $]; }
```

С помощью ключевого слова `@property` объявляется *атрибут*, называемый *свойством* (*property*). Атрибут всегда начинается со знака `@` и просто свидетельствует о том, что у определяемого символа есть определенные качества. Одни атрибуты распознаются компилятором, другие определяет и использует только сам программист[2]. В частности, атрибут

---

[1] Хотя в приведенном примере о типе аргумента `a` ничего не сказано, текущая на момент выпуска книги версия компилятора 2.057 работает указанным образом только в том случае, если `a` – массив. В ответ на пример `(7).someprop()` для функции `void someprop(int a){}` компилятор скажет, что нет свойства `someprop` для типа `int`. – *Прим. науч. ред.*

[2] Версия компилятора 2.057 не поддерживает атрибуты, объявляемые пользователем. В будущем такая поддержка может появиться. – *Прим. науч. ред.*

«property» распознается компилятором и сигнализирует о том, что функция, обладающая этим атрибутом, вызывается без () после ее имени.[1]

Также обратите внимание на использование в двух местах ключевого слова ref (см. раздел 5.2.1). Во-первых, оно употребляется при определении возвращаемого типа front; смысл в том, чтобы позволить вам изменять элементы массива, если вы того пожелаете. Во вторых, ref использует функция popFront, чтобы гарантировать непосредственное изменение среза.

Благодаря этим трем простым определениям модифицированная функция find компилируется и запускается без проблем, что доставляет огромное удовлетворение; мы обобщили функцию find так, что теперь она будет работать с любым типом, для которого определены функции empty, front и popFront, а затем завершили круг, применив обобщенную версию функции для решения той задачи, которая и послужила толчком к обобщению. Если три базовые функции для работы с T будут подвергнуты *инлайнингу* (*inlining*)[2], обобщенная версия find останется такой же эффективной, как и ее предыдущая ущербная реализация, работающая только со срезами.

Если бы функции empty, front и popFront были полезны исключительно в определении функции find, то полученная абстракция оказалась бы не особенно впечатляющей. Ладно, нам удалось применить ее к find, но пригодится ли тройка empty-front-popFront, когда мы задумаем определить другую функцию, или придется начинать все с нуля и писать другие примитивы? К счастью, обширный опыт показывает, что в понятии обобщенного доступа к коллекции данных определенно есть нечто фундаментальное. Это понятие настолько полезно, что было увековечено в виде паттерна «Итератор» в знаменитой книге «Паттерны проектирования» [27]; библиотека C++ STL [5] усовершенствовала это понятие,

---

[1] На момент выхода книги такое поведение по умолчанию носило рекомендательный характер. Функция без аргументов и без атрибута @property могла вызываться как с пустой парой скобок, так и без. Так сделано из соображений обратной совместимости с кодом, написанным до ввода данного атрибута. Заставить компилятор проверять корректность использования скобок позволяет ключ компиляции -property (dmd 2.057). В дальнейшем некорректное применение скобок может быть запрещено, поэтому там, где требуется функция, ведущая себя как свойство, следует использовать @property. – *Прим. науч. ред.*

[2] Инлайнинг (inline-подстановка) – подстановка кода функции в месте ее вызова. Позволяет снизить накладные расходы на вызов функции при передаче аргументов, переходе по адресу, обратном переходе, а также нагрузку на кэш памяти процессора. В версиях языка C до C99 это достигалось с помощью макросов, в C99 и C++ появились ключевое слово inline и inline-подстановка методов классов, описанных внутри описания класса. В языке D inline-подстановка отдается на откуп компилятору. Компилятор будет сам решать, где рационально ее применить, а где – нет. – *Прим. науч. ред.*

определив концептуальную иерархию итераторов: итератор ввода, однонаправленный итератор, двунаправленный итератор, итератор произвольного доступа.

В терминах языка D абстрактный тип данных, позволяющий перемещаться по коллекции элементов, – это *диапазон* (*range*). (Название «итератор» тоже подошло бы, но этот термин уже приобрел определенное значение в контексте ранее созданных библиотек, поэтому его использование могло бы вызвать путаницу.) У диапазонов D больше сходства с шаблоном «Итератор», чем с итераторами библиотеки STL (диапазон D можно грубо смоделировать с помощью пары итераторов из STL); тем не менее диапазоны D наследуют разбивку по категориям, определенную для итераторов STL. В частности, тройка `empty-front-popFront` определяет *диапазон ввода* (*input range*); в результате поиск хорошей реализации функции `find` привел нас к открытию сложного отношения между линейным поиском и диапазонами ввода: нельзя реализовать линейный поиск в структуре данных с меньшей функциональностью, чем у диапазона ввода, но было бы ошибкой вдруг потребовать от вашей коллекции большей функциональности, чем у диапазона ввода (например, не стоит требовать массивов с индексированным доступом к элементам). Практически идентичную реализацию функции `find` можно найти в модуле `std.algorithm` стандартной библиотеки.

5.9.2. Свести – но не к абсурду

Как насчет непростой задачи, использующей только диапазоны ввода? Условия звучат так: определить функцию `reduce`[1], которая принимает диапазон ввода `r`, операцию `fun` и начальное значение `x`, последовательно рассчитывает `x = fun(x, e)` для каждого элемента `e` из `r` и возвращает `x`. Функция высокого порядка `reduce` весьма могущественна, поскольку позволяет выразить множество интересных сверток. Эта функция – одно из основных средств многих языков программирования, позволяющих создавать функции более высокого порядка. В них она носит имена `accumulate`, `compress`, `inject`, `foldl` и т. д. Разработку функции `reduce` начнем с определения нескольких тестов модулей – в духе разработки через тестирование:

```
unittest {
   int[] r = [ 10, 14, 3, 5, 23 ];
   // Вычислить сумму всех элементов
   int sum = reduce!((a, b) { return a + b; })(0, r);
   assert(sum == 55);
   // Вычислить минимум
   int min = reduce!((a, b) { return a < b ? a : b; })(r[0], r);
   assert(min == 3);
}
```

[1] Reduce (англ.) – сокращать, сводить. – *Прим. науч. ред.*

Как можно заметить, функция reduce очень гибка и полезна – конечно, если закрыть глаза на маленький нюанс: эта функция еще не существует. Поставим цель реализовать reduce так, чтобы она работала в соответствии с определенными выше тестами. Теперь мы знаем достаточно, чтобы с самого начала написать крепкий, «промышленный» вариант функции reduce: в разделе 5.3 показано, как передать в функцию аргументы; раздел 5.4 научил нас накладывать на reduce ограничения, чтобы она принимала только осмысленные аргументы; в разделе 5.6 мы видели, как можно передать в функцию функциональные литералы через параметры-псевдонимы; а сейчас мы вплотную подошли к созданию элегантного и простого интерфейса диапазона ввода.

```
V reduce(alias fun, V, R)(V x, R range)
   if (is(typeof(x = fun(x, range.front)))
   && is(typeof(range.empty) == bool)
   && is(typeof(range.popFront())))
{
   for (; !range.empty; range.popFront()) {
      x = fun(x, range.front);
   }
   return x;
}
```

Скомпилируйте, запустите тесты модулей, и вы увидите, что все проверки пройдут прекрасно. И все же гораздо симпатичнее было бы определение reduce, где ограничения сигнатуры не достигали бы объема самой реализации. Кроме того, стоит ли писать нудные проверки, чтобы удостовериться, что R – это *диапазон ввода*? Столь многословные ограничения – это скрытое дублирование. К счастью, проверки для диапазонов уже тщательно собраны в стандартном модуле std.range, воспользовавшись которым, можно упростить реализацию reduce:

```
import std.range;

V reduce(alias fun, V, R)(V x, R range)
   if (isInputRange!R && is(typeof(x = fun(x, range.front))))
{
   for (; !range.empty; range.popFront()) {
      x = fun(x, range.front);
   }
   return x;
}
```

Такой вариант уже гораздо лучше смотрится. Имея в распоряжении функцию reduce, можно вычислить не только сумму и минимум, но и множество других агрегирующих функций, таких как число, ближайшее к заданному, наибольшее число по модулю и стандартное отклонение. Функция reduce из модуля std.algorithm стандартной библиотеки выглядит практически так же, как и наша версия выше, за исключением того, что она принимает в качестве аргументов несколько функций

для вычисления; это позволяет очень быстро вычислять значения множества агрегирующих функций, поскольку выполняется всего один проход по входным данным.

5.10. Функции с переменным числом аргументов

В традиционной программе «Hello, world!», приведенной в начале книги, для вывода приветствия в стандартный поток использовалась функция `writeln` из стандартной библиотеки. У этой функции есть интересная особенность: она принимает любое число аргументов любых типов. В языке D определить функцию с переменным числом аргументов можно разными способами, отвечающими тем или иным нуждам разработчика. Начнем с самого простого.

5.10.1. Гомогенные функции с переменным числом аргументов

Гомогенная функция с переменным числом аргументов, принимающая любое количество аргументов одного типа, определяется так:

```
import std.algorithm, std.array;

// Вычисляет среднее арифметическое множества чисел,
// переданных непосредственно или в виде массива.
double average(double[] values...) {
    if (values.empty) {
        throw new Exception("Среднее арифметическое для нуля элементов "
            "не определено");
    }
    return reduce!((a, b) { return a + b; })(0.0, values)
        / values.length;
}

unittest {
    assert(average(0) == 0);
    assert(average(1, 2) == 1.5);
    assert(average(1, 2, 3) == 2);
    // Передача массивов и срезов тоже срабатывает
    double[] v = [1, 2, 3];
    assert(average(v) == 2);
}
```

(Обратите внимание на очередное удачное использование `reduce`.) Интересная деталь функции `average`: многоточие `...` после параметра `values`, который является срезом. (Если бы это было не так или если бы параметр `values` не был последним в списке аргументов функции `average`, компилятор диагностировал бы это многоточие как ошибку.)

Вызов функции `average` со срезом массива элементов типа `double` (как показано в последней строке теста модуля) ничем не примечателен. Однако

благодаря многоточию эту функцию можно вызывать с любым числом аргументов, при условии что каждый из них можно привести к типу double. Компилятор автоматически сформирует из этих аргументов срез и передаст его в average.

Может показаться, что это средство едва ли не тот же синтаксический сахар, позволяющий компилятору заменить average(a, b, c) на average([a, b, c]). Однако благодаря своему синтаксису вызова гомогенная функция с переменным числом аргументов перегружает другие функции в своем контексте. Например:

```
// Исключительно ради аргумента
double average() {}
double average(double) {}
// Гомогенная функция с переменным числом аргументов
double average(double[] values. ) { /* То же, что и выше */     }
unittest {
   average(); // Ошибка! Двусмысленный вызов перегруженной функции!
}
```

Присутствие первых двух перегруженных версий average делает двусмысленным вызов без аргументов или с одним аргументом версии average с переменным числом аргументов. Избавиться от двусмысленности поможет явная передача среза, например average([1, 2]).

Если в одном и том же контексте одновременно присутствуют обе функции – и с фиксированным, и с переменным числом аргументов,– каждая из которых ожидает срез того же типа, что и другая, то при вызове с явно заданным срезом предпочтение отдается функции с фиксированным числом аргументов:

```
import std.stdio;

void average(double[]) { writeln("с фиксированным числом аргументов"); }
void average(double[]. ) { writeln("с переменным числом аргументов"); }
void main() {
   average(1, 2, 3);   // Пишет "с переменным числом аргументов"
   average([1, 2, 3]); // Пишет "с фиксированным числом аргументов"
}
```

Кроме срезов можно использовать в качестве аргумента массив фиксированной длины (в этом случае количество аргументов также фиксировано) и класс[1]. Подробно классы описаны в главе 6, а здесь лишь несколько слов о взаимодействии классов и функций с переменным числом аргументов.

Если написать void foo(T obj...), где T – имя класса, то внутри foo будет создан экземпляр T, причем его конструктору будут переданы аргумен-

[1] Описание этой части языка намеренно не было включено в оригинал книги, но поскольку эта возможность присутствует в текущих реализациях языка, мы добавили ее описание. – *Прим. науч. ред.*

ты, переданные функции. Если для данного набора аргументов конструктора класса T не существует, будет сгенерирована ошибка. Созданный экземпляр является локальным для данной функции, память под него может быть выделена в стеке, поэтому он не возвращается функцией.

5.10.2. Гетерогенные функции с переменным числом аргументов

Вернемся к функции writeln. Она явно должна делать не совсем то же самое, что функция average, поскольку writeln принимает аргументы разных типов. Для обработки произвольного числа аргументов произвольных типов предназначена гетерогенная функция с переменным числом аргументов, которую определяют так:

```
import std.conv;

void writeln(T...)(T args) {
   foreach (arg; args) {
      stdout.rawWrite(to!string(arg));
   }
   stdout.rawWrite('\n');
   stdout.flush();
}
```

Эта реализация немного сырovата и неэффективна, но она работает. T внутри writeln – ***кортеж типов параметров*** (тип, который группирует несколько типов), а args – ***кортеж параметров***. Цикл foreach определяет, что args – это кортеж типов, и генерирует код, радикально отличающийся от того, что получается в результате обычного выполнения инструкции foreach (например, когда цикл foreach применяется для просмотра массива). Рассмотрим, например, такой вызов:

```
writeln("Печатаю целое: ", 42, " и массив: ", [1, 2, 3]);
```

Для такого вызова конструкция foreach сгенерирует код следующего вида:

```
// Аппроксимация сгенерированного кода
void writeln(string a0, int a1, string a2, int[] a3) {
   stdout.rawWrite(to!string(a0));
   stdout.rawWrite(to!string(a1));
   stdout.rawWrite(to!string(a2));
   stdout.rawWrite(to!string(a3));
   stdout.rawWrite('\n');
   stdout.flush();
}
```

В модуле std.conv определены версии to!string для всех типов (включая и сам тип string, для которого функция to!string – тождественное отображение), так что функция работает, по очереди преобразуя каждый аргумент в строку и печатая ее «сырые» байты в стандартный поток вывода.

Обратиться к типам или значениям кортежа параметров можно и без цикла foreach. Если n – известное во время компиляции неизменяемое число, то выражение T[n] возвратит n-й тип, а выражение args[n] – n-е значение в кортеже параметров. Получить число аргументов можно с помощью выражения T.length или args.length (оба являются константами, известными во время компиляции). Если вы уже заметили сходство с массивами, то не будете удивлены, узнав, что с помощью выражения T[$ - 1] можно получить доступ к последнему типу в T (а args[$ - 1] – псевдоним для последнего значения в args). Например:

```
import std.stdio;

void testing(T...)(T values) {
   writeln("Переданных аргументов: ", values.length, ":");
   // Обращение к каждому индексу и каждому значению
   foreach (i, value; values) {
      writeln(i, ": ", typeid(T[i]), " ", value);
   }
}
void main() {
   testing(5, "здравствуй", 4.2);
}
```

Эта программа напечатает:

```
Переданных аргументов: 3.
0: int 5
1: immutable(char)[] здравствуй
2: double 4.2
```

5.10.2.1. Тип без имени

Функция writeln делает слишком много специфичного, чтобы быть обобщенной: она всегда добавляет в конце '\n' и затем использует функцию flush для записи данных буферов потока. Попробуем определить функцию writeln через базовую функцию write, которая просто выводит все аргументы по очереди:

```
import std.conv;

void write(T...)(T args) {
   foreach (arg; args) {
      stdout.rawWrite(to!string(arg));
   }
}

void writeln(T...)(T args) {
   write(args, '\n');
   stdout.flush();
}
```

Обратите внимание, как writeln делегирует запись args и '\n' функции write. При передаче кортеж параметров автоматически разворачивается, так что вызов writeln(1, "2", 3) делегирует функции write запись из четырех, а не трех аргументов. Такое поведение немного необычно и не совсем понятно, поскольку практически во всех остальных случаях в D под одним идентификатором понимается одно значение. Этот пример может удивить даже подготовленных:

```
void fun(T...)(T args) {
   gun(args);
}

void gun(T)(T value) {
   writeln(value);
}

unittest {
   fun(1);       // Все в порядке
   fun(1, 2.2); // Ошибка! Невозможно найти функцию gun
                // принимающую два аргумента!
}
```

Первый вызов проходит гладко, чего нельзя сказать о втором. Вы ожидали, что все будет в порядке, ведь любое значение (а значит, и args) обладает каким-то типом, и потому тип args должен выводиться функцией gun. Но что происходит на самом деле?

Все значения действительно обладают типами, которые корректно отслеживаются компилятором. Виновен вызов gun(args), поскольку компилятор автоматически расширяет этот вызов, когда бы кортеж параметров ни передавался в качестве аргумента функции. Даже если вы написали gun(args), компилятор всегда развернет такой вызов до gun(args[0], args[1], ..., args[$ - 1]). Под вторым вызовом подразумевается вызов gun(args[0], args[1]), который требует несуществующей функции gun с двумя аргументами, – отсюда и ошибка.

Чтобы более глубоко исследовать этот случай, напишем «забавную» функцию fun для печати типа значения args.

```
void fun(T...)(T args) {
   writeln(typeof(args).stringof);
}
```

Конструкция typeof – не вызов функции; это выражение всего лишь возвращает тип args, поэтому можно не волноваться относительно автоматической развертки. Свойство .stringof, присущее всем типам, возвращает имя типа, так что давайте еще раз скомпилируем и запустим программу. Она печатает:

```
(int)
(int, double)
```

Итак, действительно похоже на то, что компилятор отслеживает типы кортежей параметров, и для них определено строковое представление. Тем не менее невозможно явно определить кортеж параметров: типа (int, double) не существует.

```
// Бесполезно
(int, double) value = (1, 4.2);
```

Все объясняется тем, что кортежи в своем роде уникальны: это типы, которые внутренне используются компилятором, но не могут быть выражены в тексте программы. Никаким образом невозможно взять и написать тип кортежа параметров. Потому нет и литерала, порождающего вывод кортежа параметров (если бы был, то необходимость в указании имени типа отпала бы: ведь есть ключевое слово auto).

5.10.2.2. Тип данных Tuple и функция tuple

Концепция типов без имен и значений без литералов может заинтересовать любителя острых ощущений, однако программист практического склада увидит здесь нечто угрожающее. К счастью (наконец-то! эти слова должны были появиться рано или поздно), это не столько ограничение, сколько способ сэкономить на синтаксисе. Есть замечательная возможность представлять типы кортежей параметров с помощью типа Tuple, а значения кортежей параметров – с помощью функции tuple. И то и другое находится в стандартном модуле std.typecons. Таким образом, кортеж параметров, содержащий int и double, можно записать так:

```
import std.typecons;
unittest {
   Tuple!(int, double) value = tuple(1, 4.2); // Ого!
}
```

Учитывая, что выражение tuple(1, 4.2) возвращает значение типа Tuple!(int, double), следующий код эквивалентен только что представленному:

```
auto value = tuple(1, 4.2); // Двойное "ого!"
```

Тип Tuple!(int, double) такой же, как и все остальные типы, он не делает никаких фокусов с автоматической разверткой, так что если вы хотите развернуть его до составных частей, нужно сделать это явно с помощью свойства .expand типа Tuple. Для примера переплавим нашу программу с функциями fun и gun и в результате получим следующий код:

```
import std.stdio, std.typecons;

void fun(T...)(T args) {
  // Создать кортеж, чтобы "упаковать" все аргументы в одно значение
   gun(tuple(args));
}

void gun(T)(T value) {
```

```
    // Расширить кортеж и получить исходное множество параметров
    writeln(value.expand);
}

void main() {
    fun(1);       // Все в порядке
    fun(1, 2.2); // Все в порядке
}
```

Посмотрите, как функция fun группирует все аргументы в один кортеж (Tuple) и передает его в функцию gun, которая разворачивает полученный кортеж, извлекая все, что он содержит. Выражение value.expand автоматически заменяется на список аргументов, содержащий все, что вы отправили в Tuple.

В реализации типа Tuple есть пара тонких моментов, но она использует средства, доступные любому программисту. Изучение определения типа Tuple (которое можно найти в стандартной библиотеке) было бы полезным упражнением.

5.10.3. Гетерогенные функции с переменным числом аргументов. Альтернативный подход[1]

Предыдущий подход всем хорош, однако применение шаблонов накладывает на функции ряд ограничений. Поскольку приведенная выше реализация использует шаблоны, для каждого возможного кортежа параметров создается свой экземпляр шаблонной функции. Это не позволяет делать шаблонные функции виртуальными методами класса, объявлять их нефинальными членами интерфейсов, а при невнимательном подходе может приводить к излишнему разрастанию результирующего кода (поэтому шаблонная функция должна быть небольшой, чтобы компилятор счел возможной ее inline-подстановку). Поэтому D предлагает еще два способа объявить функцию с переменным числом аргументов. Оба способа были добавлены в язык до появления шаблонов с переменным числом аргументов, и сегодня считаются небезопасными и устаревшими. Тем не менее они присутствуют и используются в текущих реализациях языка, чаще всего из соображений совместимости.

5.10.3.1. Функции с переменным числом аргументов в стиле C

Первый способ язык D унаследовал от языка C. Вспомним функцию printf. Вот ее сигнатура на D:

```
extern(C) int printf(in char* format, ...);
```

[1] Описание этой части языка намеренно не было включено в оригинал книги, но поскольку эта возможность присутствует в текущих реализациях языка, мы добавили ее описание в перевод. – *Прим. науч. ред.*

Разберем ее по порядку. Запись extern(C) обозначает тип компоновки. В данном случае указано, что функция использует тип компоновки C. То есть параметры передаются в функцию в соответствии с соглашением о вызовах языка C. Также в C не используется искажение имен (mangling) функций, поэтому такая функция не может быть перегружена по типам аргументов. Если две такие функции с одинаковыми именами объявлены в разных модулях, возникнет конфликт имен. Как правило, extern(C) используется для взаимодействия с кодом, уже написанным на C или других языках. in char* format – обязательный первый аргумент функции, за которым следует переменное число аргументов, что символизирует уже знакомое нам многоточие (...).

Для начала вспомним, как аргументы передаются функции в языке C. C передает аргументы через стек, помещая в него аргументы, начиная с последнего. За удаление аргументов из стека отвечает вызывающая подпрограмма. Например, при вызове printf("%d + %d = %d", 2, 3, 5) первым в стек будет помещен аргумент 5, после него 3, затем 2 и последней – строка формата. В итоге строка формата оказывается на вершине стека и будет доступна в вызываемой функции. Для получения остальных аргументов в C используются макросы, определенные в файле stdarg.h.

В языке D соответствующие функции определены в модуле std.c.stdarg. Во-первых, в данном модуле определен тип va_list, который является указателем на список необязательных аргументов. Функция va_start инициализирует переменную va_list указателем на начало списка необязательных аргументов.

```
void va_start(T)( out va_list ap, ref T parmn );
```

Первый аргумент – инициализируемая переменная va_list, второй – ссылка на последний обязательный аргумент, то есть последний аргумент, тип которого известен. На основании него вычисляется указатель на первый элемент списка необязательных аргументов. Именно поэтому функция с переменным числом аргументов в C должна иметь хотя бы один обязательный параметр, чтобы va_start было к чему привязаться. Объявление extern(C) int foo(...); недопустимо.

Функция va_arg получает значение очередного аргумента заданного типа. Тип этого аргумента может быть получен в результате каких-то операций с предыдущими аргументами, и проверить правильность его получения невозможно. Указатель на список при этом изменяется так, чтобы он указывал на следующий элемент списка.

```
T va_arg(T)( ref va_list ap );
```

Функция va_copy предназначена для копирования переменной типа va_list. Если va_list – указатель на стек функции, выполняется копирование указателя. Если же в вашей системе аргументы передаются через регистры, потребуется выделение памяти и копирование списка.

```
void va_copy( out va_list dest, va_list src );
```

Функция va_end вызывается по завершении работы со списком аргументов. Каждый вызов va_start или va_copy должен сопровождаться вызовом va_end.

```
void va_end( va_list ap );
```

Интерфейс stdarg является кроссплатформенным, а сама реализация функций с переменным числом аргументов может быть различной для разных платформ. В некоторых платформах аргументы передаются через стек, и va_list – указатель на верхний элемент списка в стеке. В некоторых аргументы могут передаваться через регистры. Также разным может быть выравнивание элементов в стеке и направление роста стека. Поэтому следует пользоваться именно этим интерфейсом, а не пытаться договориться с функцией в обход него. Пример функции для преобразования в строку значения нужного типа:

```
import std.c.stdarg, std.conv;

extern(C) string cToString(string type,   .) {
   va_list args_list;
   va_start(args_list, type);
   scope(exit) va_end(args_list);
   switch (type) {
      case "int":
         auto int_val = va_arg!int(args_list);
         return to!string(int_val);
      case "double":
         auto double_val = va_arg!double(args_list);
         return to!string(double_val);
      case "complex":
         auto re_val = va_arg!double(args_list);
         auto im_val = va_arg!double(args_list);
         return to!string(re_val) ~ " + " ~ to!string(im_val) ~ "i";
      case "string":
         return va_arg!string(args_list);
      default:
         assert(0, "Незнакомый тип");
   }
}

unittest {
   assert(cToString("int"  5) == "5");
   assert(cToString("double"  2.0) == "2");
   assert(cToString("string"  "Test string") == "Test string");
   assert(cToString("complex"  3.5, 2.7) == "3.5 + 2.7i");
}
```

В этом примере мы первым аргументом передаем тип следующих аргументов, и на основании этого аргумента функция определяет, каких аргументов ей ждать дальше. Однако если мы допустим ошибку в вызове, то спасти нас уже никто не сможет. В этом и заключается опасность

подобных функций: ошибка в вызове может привести к аппаратной ошибке внутри самой функции. Например, если мы напишем:

```
cToString("string" 3.5, 2.7);
```

результат будет непредсказуемым. Поэтому, например, функция scanf может оказаться небезопасной, если строка формата берется из ненадежного источника, ведь с правильно подобранной строкой формата и аргументом можно получить перезапись адреса возврата функции и заставить программу выполнить какой-то свой, наверняка вредоносный код. Поэтому язык D предлагает менее опасный способ создания функций с переменным числом аргументов.

5.10.3.2. Функции с переменным числом аргументов в стиле D

Функцию с переменным числом аргументов в стиле D можно объявить так:

```
void foo(...);
```

То есть делается абсолютно то же самое, что и в случае выше, но выбирается тип компоновки D (по умолчанию или явным указанием extern(D)), и обязательный аргумент можно не указывать. В самой же приведенной функции применяется не такой подход, как в языке C. Внутри такой функции доступны два идентификатора: _arguments типа TypeInfo[] и _argptr типа va_list. Идентификатор _argptr указывает на начало списка аргументов, а _arguments – на массив идентификаторов типа для каждого переданного аргумента. Количество переданных аргументов соответствует длине массива.

Об идентификаторах типов следует рассказать подробнее. Идентификатор типа – это объект класса TypeInfo или производного от него. Получить идентификатор типа T можно с помощью выражения typeid(T). Для каждого типа есть один и только один идентификатор. То есть равенство typeid(int) is typeid(int) всегда верно. Полный список параметров класса TypeInfo следует искать в документации по вашему компилятору или в модуле object. Модуль object, объявленный в файле object.di, импортируется в любом модуле по умолчанию, то есть можно использовать любые объявленные в нем символы без каких-то дополнительных объявлений. Вот безопасный вариант предыдущего примера:

```
import std.c.stdarg, std.conv;

string dToString(string type, ...) {
   va_list args_list;
   va_copy(args_list, _argptr);
   scope(exit) va_end(args_list);
   switch (type) {
      case "int":
         assert(_arguments.length == 1 && _arguments[0] is typeid(int),
            "Аргумент должен иметь тип int.");
```

```
            auto int_val = va_arg!int(args_list);
            return to!string(int_val);
        case "double":
            assert(_arguments.length == 1 &&_arguments[0] is typeid(double),
                "Аргумент должен иметь тип double.");
            auto double_val = va_arg!double(args_list);
            return to!string(double_val);
        case "complex":
            assert(_arguments.length == 2 &&
                _arguments[0] is typeid(double) &&
                _arguments[1] is typeid(double),
                "Для типа complex должны быть переданы " ~
                "два аргумента типа double.");
            auto re_val = va_arg!double(args_list);
            auto im_val = va_arg!double(args_list);
            return to!string(re_val) ~ " + " ~ to!string(im_val) ~ "i";
        case "string":
            assert(_arguments.length == 1 &&_arguments[0] is typeid(string),
                "Аргумент должен иметь тип string.");
            return va_arg!string(args_list).idup;
        default:
            assert(0);
    }
}

unittest {
    assert(dToString("int"  5) == "5");
    assert(dToString("double"  2.0) == "2");
    assert(dToString("string"  "Test string") == "Test string");
    assert(dToString("complex"  3.5, 2.7) == "3.5 + 2.7i");
}
```

Этот вариант автоматически проверят типы переданных аргументов. Однако не забывайте, что корректность типа, переданного va_arg**, остается за вами – использование неправильного типа приведет к непредсказуемой ситуации. Если вас это беспокоит, то для полной безопасности вы можете использовать конструкцию** Variant **из модуля стандартной библиотеки** std.variant**:**

```
import std.stdio, std.variant;

void pseudoVariadic(Variant[] vars) {
    foreach (var; vars)
        if (var.type == typeid(string))
            writeln("Строка: "  var.get!string);
        else
        if (var.type == typeid(int))
            writeln("Целое число: "  var.get!int);
        else
            writeln("Незнакомый тип: "  var.type);
}
```

```
void templatedVariadic(T. .)(T args) {
   pseudoVariadic(variantArray(args));
}

void main() {
   templatedVariadic("Здравствуй, мир!"  42);
}
```

При этом функция `templatedVariadic`, скорее всего, будет встроена в код путем inline-подстановки, и накладных расходов на лишний вызов функции и разрастание шаблонного кода не будет.

5.11. Атрибуты функций

К функциям на D можно присоединять *атрибуты* – особые средства, извещающие программиста и компилятор о том, что функция обладает некоторыми качествами. Функции проверяются на соответствие своим атрибутам, поэтому, чтобы узнать важную информацию о поведении функции, достаточно взглянуть на ее сигнатуру: атрибуты предоставляют твердые гарантии, это не простые комментарии или соглашения.

5.11.1. Чистые функции

Чистота функций – заимствованное из математики понятие, полезное как в теории, так и на практике. В языке D функция считается чистой, если все, что она делает, сводится к возвращению результата и возвращаемое значение зависит только от ее аргументов.

В классической математике все функции чистые, поскольку в классической математике нет состояний и изменений. Чему равен $\sqrt{2}$? Примерно 1,4142; так было вчера, будет завтра и вообще всегда. Можно доказать, что значение $\sqrt{2}$ было тем же еще до того, как человечество открыло корни, алгебру, числа, и даже *до* появления человечества, способного оценить красоту математики, и столь же долго пребудет неизменным после тепловой смерти Вселенной. Математические результаты вечны.

Чистота – это благо для функций, пусть даже иногда и с ограничениями, впрочем, как и в жизни. (Кстати, как и в жизни, чистоты не так просто достичь. Более того, по мнению некоторых, излишества в некоторых проявлениях чистоты на самом деле могут раздражать.) В пользу чистоты говорит тот факт, что о чистой функции легче делать выводы. Чистота гарантирует: чтобы узнать, что делает та или иная функция, достаточно взглянуть на ее вызов. Можно заменять эквивалентные вызовы функций значениями, а значения – эквивалентными вызовами функций. Можно быть уверенным, что ошибки в чистых функциях не обладают эффектом шрапнели – они не могут повлиять на что-либо еще помимо результата самой функции.

Кроме того, чистые функции могут выполняться в буквальном смысле параллельно, так как они никаким образом, кроме их результата, не взаимодействуют с остальным кодом программы. В противоположность им, насыщенные изменениями[1] нечистые функции при параллельном выполнении склонны наступать друг другу на пятки. Но даже если выполнять их последовательно, результат может неуловимо зависеть от порядка, в котором они вызываются. Многих из нас это не удивляет — мы настолько свыклись с таким раскладом, что считаем преодоление трудностей неотъемлемой частью процесса написания кода. Но если хотя бы некоторые части приложения будут написаны «чисто», это принесет большую пользу, освежив программу в целом.

Определить чистую функцию можно, добавив в начало ее определения ключевое слово pure:

```
pure bool leapYear(uint y) {
   return (y % 4) == 0 && (y % 100 || (y % 400) == 0);
}
```

Например, сигнатура функции

```
pure bool leapYear(uint y);
```

гарантирует пользователю, что функция leapYear не пишет в стандартный поток вывода. Кроме того, уже по сигнатуре видно, что вызов leapYear(2020) всегда будет возвращать одно и то же значение.

Компилятор также в курсе значения ключевого слова pure, и именно он ограждает программиста от любых действий, способных нарушить чистоту функции leapYear. Приглядитесь к следующим изменениям:

```
pure bool leapYear(uint y) {
   auto result = (y % 4) == 0 && (y % 100 || (y % 400) == 0);
   if (result) writeln(y, " - високосный год!"); // Ошибка!
      // Из чистой функции невозможно вызвать нечистую функцию!
   return result;
}
```

Функция writeln не является и не может стать чистой. И если бы она заявляла обратное, компилятор бы избавил ее от такого заблуждения. Компилятор гарантирует, что чистая функция вызывает только чистые функции. Вот почему измененная функция leapYear не компилируется. С другой стороны, проверку компилятора успешно проходят такие функции, как daysInYear:

```
// Чистота подтверждена компилятором
pure uint daysInYear(uint y) {
   return 365 + leapYear(y);
}
```

[1] В данном контексте речь идет об изменениях, которые повлияли бы на последующие вызовы функции, например об изменении глобальных переменных. — *Прим. науч. ред.*

5.11.1.1. «Чист тот, кто чисто поступает»

По традиции функциональные языки запрещают абсолютно любые изменения, чтобы программа могла называться чистой. D ослабляет это ограничение, разрешая функциям изменять собственное локальное и временное состояние. Таким образом, даже если внутри функции есть изменения, для окружающего кода она все еще непогрешима.

Посмотрим, как работает это допущение. В качестве примера возьмем наивную реализацию функции Фибоначчи в функциональном стиле:

```
ulong fib(uint n) {
   return n < 2 ? n : fib(n   1) + fib(n   2);
}
```

Ни один преподаватель программирования никогда не должен учить реализовывать расчет чисел Фибоначчи таким способом. Чтобы вычислить результат, функции `fib` требуется *экспоненциальное время*, поэтому все, чему она может научить, – это пренебрежение сложностью и ценой вычислений, лозунг «небрежно, зато находчиво» и спортивный стиль вождения. Хотите знать, чем плох экспоненциальный порядок? Вызовы `fib(10)` и `fib(20)` на современной машине не займут много времени, но вызов `fib(50)` обрабатывается уже 19 минут. Вполне вероятно, что вычисление `fib(1000)` переживет человечество (только смысла в этом никакого, в отличие от примера с $\sqrt{2}$.)

Хорошо, но как выглядит «правильная» функциональная реализация Фибоначчи?

```
ulong fib(uint n) {
   ulong iter(uint i, ulong fib_1, ulong fib_2) {
      return i == n
         ? fib_2
         : iter(i + 1, fib_1 + fib_2, fib_1);
   }
   return iter(0, 1, 0);
}
```

Переработанная версия вычисляет `fib(50)` практически мгновенно. Эта реализация требует для выполнения $O(n)$[1] времени, поскольку оптимизация хвостовой рекурсии (см. раздел 1.4.2) позволяет уменьшить сложность вычислений. (Стоит отметить, что для расчета чисел Фибоначчи существуют и алгоритмы с временем выполнения $O(\log n)$).

Проблема в том, что новая функция `fib` как бы утратила былое великолепие. Особенность переработанной реализации – две переменные состояния, маскирующиеся под параметры функции, и вполне можно было с чистой совестью написать явный цикл, который зачем-то был закамуфлирован функцией `iter`:

---

[1] «О» большое – математическое обозначение, применяемое при оценке асимптотической сложности алгоритма. – *Прим. ред.*

```
ulong fib(uint n) {
   ulong fib_1 = 1, fib_2 = 0;
   foreach (i; 0    n) {
      auto t = fib_1;
      fib_1 += fib_2;
      fib_2 = t;
   }
   return fib_2;
}
```

К сожалению, это уже не функциональный стиль. Только посмотрите на все эти изменения, происходящие в цикле. Один неверный шаг – и с вершин математической чистоты мы скатились к неискушенности чумазых низов.

Но подумав немного, мы увидим, что итеративная функция `fib` *не* такая уж чумазая. Если принять ее за черный ящик, то можно заметить, что при одних и тех же аргументах функция `fib` всегда возвращает один и тот же результат, а ведь «красив тот, кто красиво поступает». Тот факт, что она использует локальное изменение состояния, делает ее менее функциональной по букве, но не по духу. Продолжая эту мысль, приходим к очень интересному выводу: пока изменяемое состояние внутри функции остается полностью *временным* (то есть хранит данные в стеке) и *локальным* (то есть не передается по ссылке другим функциям, которые могут его нарушить), эту функцию можно считать чистой.

Вот как D определяет функциональную чистоту: в реализации чистой функции разрешается использовать изменения, если они временные и локальные. Сигнатуру такой функции можно снабдить ключевым словом `pure`, и компилятор без помех скомпилирует этот код:

```
pure ulong fib(uint n) {
      // Итеративная реализация
}
```

Принятые в D допущения, смягчающие математическое понятие чистоты, очень полезны, поскольку позволяют взять лучшее из двух миров: железные гарантии функциональной чистоты и удобную реализацию (если код с изменениями более предпочтителен).

5.11.2. Атрибут nothrow

Атрибут `nothrow` сообщает, что данная функция никогда не порождает исключения. Как и атрибут `pure`, атрибут `nothrow` проверяется во время компиляции. Например:

```
import std.stdio;

nothrow void tryLog(string msg) {
   try {
      stderr.writeln(msg);
```

```
    } catch (Exception) {
        // Проигнорировать исключение
    }
}
```

Функция tryLog прилагает максимум усилий, чтобы записать в журнал сообщение. Если возникает исключение, она его молча игнорирует. Это качество позволяет использовать функцию tryLog на критических участках кода. При определенных обстоятельствах было бы глупо позволить некоторой важной транзакции сорваться только из-за невозможности сделать запись в журнал. Устройство кода, представляющего собой транзакцию, основано на том, что некоторые из его участков никогда не порождают исключения, а применение атрибута nothrow позволяет статически гарантировать это свойство критических участков.

Проверка семантики функций с атрибутом nothrow гарантирует, что исключение никогда не просочится из функции. Для каждой инструкции внутри функции должно быть истинно одно из утверждений: 1) эта инструкция не порождает исключения (в случае вызова функции это возможно, только если вызываемая функция также не порождает исключения), 2) эта инструкция расположена внутри инструкции try, «съедающей» исключения. Проиллюстрируем второй случай примером:

```
nothrow void sensitive(Widget w) {
    tryLog("Начинаем опасную операцию");
    try {
        w.mayThrow(); // Вызов может породить исключение
        tryLog("Опасная операция успешно завершена");
    } catch (Exception) {
        tryLog("Опасная операция завершилась неудачей");
    }
}
```

Первый вызов функции tryLog можно не помещать в блок try, поскольку компилятор уже знает, что эта функция не порождает исключения. Аналогично вызов внутри блока catch можно не «защищать» с помощью дополнительного блока try.

Как соотносятся атрибуты pure и nothrow? Может показаться, что они совершенно независимы друг от друга, но на самом деле между ними есть некоторая взаимосвязь. По крайней мере в стандартной библиотеке многие функции, например самые трансцендентные (такие как exp, sin, cos), имеют оба атрибута – и pure, и nothrow.

5.12. Вычисления во время компиляции

В подтверждение поговорки, что счастье приходит к тому, кто умеет ждать (или терпеливо читать), в этом последнем разделе обсуждается очень интересное средство D. Лучшее в этом средстве то, что вам не нужно много учиться, чтобы начать широко его применять.

Рассмотрим пример, достаточно большой, чтобы быть осмысленным. Предположим, вы хотите создать лучшую библиотеку генераторов случайных чисел. Есть много разных генераторов случайных чисел, в том числе линейные конгруэнтные генераторы [35, раздел 3.2.1, с. 10–26]. У таких генераторов есть три целочисленных параметра: модуль $m > 0$, множитель $0 < a < m$ и наращиваемое значение[1] $0 < c < m$. Начав с произвольного начального значения $0 \le x_0 < m$, линейный конгруэнтный генератор вычисляет псевдослучайные числа по следующей рекуррентной формуле:

$$x_{n+1} = (ax_n + c) \bmod m$$

Запрограммировать такой алгоритм очень просто: достаточно сохранять состояние, определяемое числами m, a, c и x_n, и определить функцию `getNext` для получения следующего значения x_{n+1}.

Но здесь есть подвох. Не все комбинации a, m и c дадут хороший генератор случайных чисел. Для начала, при $a = 1$ и $c = 1$ генератор формирует последовательность 0, 1, ..., $m - 1$, 0, ..., $m - 1$, 0, 1, ..., которую случайной уж никак не назовешь.

С бóльшими значениями a и c таких очевидных рисков можно избежать, однако появляется менее заметная проблема: периодичность. Из-за оператора деления по модулю числа генерируются всегда между 0 и $m - 1$, так что неплохо было бы сделать значение m настолько большим, насколько это возможно (обычно в качестве значения этого параметра берут степень двойки, чтобы оно соответствовало размеру машинного слова: это позволяет обойтись без затрат на деление по модулю). Проблема в том, что сгенерированная последовательность может обладать периодом гораздо меньшим, чем m. Пусть мы работаем с типом `uint` и выбираем $m = 2^{32}$ (тогда нам даже операция деления по модулю не нужна), $a = 210$, $c = 123$, а для x_0 возьмем какое-нибудь сумасшедшее значение, например 1780588661. Запустим следующую программу:

```
import std.stdio;

void main() {
   enum uint a = 210, c = 123, x0 = 1_780_588_661;
   auto x = x0;
   foreach (i; 0    100) {
      x = a * x + c;
      writeln(x);
   }
}
```

Вместо пестрого набора случайных чисел мы увидим нечто неожиданное:

[1] Равенство c нулю также допустимо, но соответствующая теоретическая часть гораздо сложнее, потому ограничимся значениями $c > 0$.

```
1     261464181
2     3367870581
3     2878185589
4     3123552373
5     3110969461
6     468557941
7     3907887221
8     317562997
9     2263720053
10    2934808693
11    2129502325
12    518889589
13    1592631413
14    3740115061
15    3740115061
16    3740115061
17
```

Начинает генератор вполне задорно. По крайней мере, с непривычки может показаться, что он неплохо справляется с генерацией случайных чисел. Однако уже с 14-го шага генератор зацикливается: по странному стечению обстоятельств, породить которое могла только математика, 3740115061 оказалось (и всегда будет оказываться) точно равным $(3740115061 * 210 + 123) \bmod 2^{32}$. Это период единицы, худшее из возможного!

Значит, необходимо выбрать такие параметры m, a и c, чтобы сгенерированная последовательность псевдослучайных чисел гарантированно имела большой период. Дальнейшие исследования этой проблемы выявили следующие условия генерации последовательности псевдослучайных чисел с периодом m (наибольший возможный период):

1. c и m взаимно просты.
2. Значение $a - 1$ кратно всем простым делителям m.
3. Если $a - 1$ кратно 4, то и m кратно 4.

Взаимную простоту c и m можно легко проверить сравнением наибольшего общего делителя этих чисел с 1. Для вычисления наибольшего общего делителя воспользуемся алгоритмом Евклида[1]:

```
// Реализация алгоритма Евклида
ulong gcd(ulong a, ulong b) {
   while (b) {
      auto t = b;
      b = a % b;
      a = t;
   }
   return a;
}
```

[1] Непонятно как, но алгоритм Евклида всегда умудряется попадать в хорошие (хм...) книги по программированию.

Евклид выразил свой алгоритм с помощью вычитания, а не деления по модулю. Для версии с делением по модулю требуется меньше итераций, но на современных машинах % может вычисляться довольно-таки медленно (видимо, именно это и остановило Евклида).

Реализовать вторую проверку немного сложнее. Можно было бы написать функцию `factorize`, возвращающую все возможные простые делители числа с их степенями, и воспользоваться ею, но `factorize` – это больше, чем нам необходимо. Стремясь к простейшему решению, которое могло бы сработать, проще всего написать функцию `primeFactorsOnly(n)`, возвращающую произведение простых делителей `n`, но без степеней. Тогда наша задача сводится к проверке выражения `(a - 1) % primeFactorsOnly(m) == 0`. Итак, приступим к реализации функции `primeFactorsOnly`.

Есть много способов получить простые делители некоторого числа n. Один из простых: сгенерировать простые числа p_1, p_2, p_3, ..., для каждого значения p_k выяснить, делится ли n на p_k, и если делится, то умножить p_k на значение-аккумулятор r. Когда очередное число p_k окажется больше n, вычисления прекращаются. Аккумулятор r содержит искомое значение – произведение всех простых делителей n, взятых по одному разу.

(Догадываюсь, что сейчас вы задаетесь вопросом, имеет ли все это отношение к вычислениям во время компиляции. Ответ: имеет. Прошу немного терпения.)

Более простую версию можно получить, избавившись от генерации простых чисел. Можно просто вычислять n mod k для возрастающих значений k, образующих следующую последовательность (начиная с 2): 2, 3, 5, 7, 9, ... Всякий раз, когда n делится на k, аккумулятор умножается на k, а n «очищается» от всех степеней k: n присваивается значение n / k, пока n делится на k. Таким образом, мы сохранили значение k и одновременно уменьшили число n настолько, что теперь оно не делится на k. Это не выглядит как самый экономный метод, но задумайтесь о том, что генерация простых чисел могла бы потребовать сравнимых трудозатрат, по крайней мере в случае простой реализации. Реализация этой идеи могла бы выглядеть так:

```
ulong primeFactorsOnly(ulong n) {
   ulong accum = 1;
   ulong iter = 2;
   for (; n >= iter * iter; iter += 2 - (iter == 2)) {
      if (n % iter) continue;
      accum *= iter;
      do n /= iter; while (n % iter == 0);
   }
   return accum * n;
}
```

Команда `iter += 2 - (iter == 2)`, обновляющая значение переменной `iter`, всегда увеличивает его на 2, кроме случая, когда `iter` равно 2: тогда значение этой переменной заменяется на 3. Таким образом, переменная `iter`

принимает значения 2, 3, 5, 7, 9 и т. д. Было бы слишком расточительно проверять каждое четное число, например 4, поскольку число 2 уже было проверено и все его степени извлечены из n.

Почему в качестве условия продолжения цикла выбрана проверка n >= iter * iter, а не n >= iter? Ответ не вполне прямолинеен. Если число iter больше $\sqrt{n}$ и отличается от самого числа n, то есть уверенность, что число n не делится на число iter: если бы делилось, должен был бы существовать некоторый множитель k, такой, что n == k * iter, но все делители меньше iter только что были рассмотрены, так что k должно быть больше iter, и следовательно, произведение k * iter — больше n, что делает равенство невозможным.

Протестируем функцию primeFactorsOnly:

```
unittest {
   assert(primeFactorsOnly(100) == 10);
   assert(primeFactorsOnly(11) == 11);
   assert(primeFactorsOnly(7 * 7 * 11 * 11 * 15) == 7 * 11 * 15);
   assert(primeFactorsOnly(129 * 2) == 129 * 2);
}
```

В завершение нам необходима небольшая функция-обертка, выполняющая три рассмотренные проверки трех потенциальных параметров линейного конгруэнтного генератора:

```
bool properLinearCongruentialParameters(ulong m, ulong a, ulong c) {
   // Проверка границ
   if (m == 0 || a == 0 || a >= m || c == 0 || c >= m) return false;
   // c и m взаимно просты
   if (gcd(c, m) != 1) return false;
   // Значение a - 1 кратно всем простым делителям m
   if ((a - 1) % primeFactorsOnly(m)) return false;
   // Если a - 1 кратно 4, то и m кратно 4
   if ((a - 1) % 4 == 0 && m % 4) return false;
   // Все тесты пройдены
   return true;
}
```

Протестируем некоторые популярные значения m, a и c:

```
unittest {
   // Наш неподходящий пример
   assert(!properLinearCongruentialParameters(
      1UL << 32, 210, 123));
   // Пример из книги "Numerical Recipes" [48]
   assert(properLinearCongruentialParameters(
      1UL << 32, 1664525, 1013904223));
   // Компилятор Borland C/C++
   assert(properLinearCongruentialParameters(
      1UL << 32, 22695477, 1));
   // glibc
   assert(properLinearCongruentialParameters(
```

```
        1UL << 32, 1103515245, 12345));
    // ANSI C
    assert(properLinearCongruentialParameters(
        1UL << 32, 134775813, 1));
    // Microsoft Visual C/C++
    assert(properLinearCongruentialParameters(
        1UL << 32, 214013, 2531011));
}
```

Похоже, функция properLinearCongruentialParameters работает как надо, то есть мы справились со всеми деталями тестирования состоятельности линейного конгруэнтного генератора. Так что пора притормозить, заглушить мотор и покаяться. Какое отношение имеет вся эта простота и делимость к вычислениям во время компиляции? Где мясо?[1] Где шаблоны, макросы или как там они еще называются? Многообещающие инструкции static if? Умопомрачительные генерация кода и расширение кода?

На самом деле, вы только что увидели все, что только можно рассказать о вычислениях во время компиляции. Задав константам m, n и c любые числовые значения, можно вычислить properLinearCongruentialParameters *во время компиляции*, никак не изменяя эту функцию или функции, которые она вызывает. В компилятор D встроен интерпретатор, который вычисляет функции на D во время компиляции – со всей арифметикой, циклами, изменениями, ранними возвратами и даже трансцендентными функциями.

От вас требуется только указать компилятору, что вычисления нужно выполнить во время компиляции. Для этого есть несколько способов:

```
unittest {
    enum ulong m = 1UL << 32, a = 1664525, c = 1013904223;
    // Способ 1: воспользоваться инструкцией static assert
    static assert(properLinearCongruentialParameters(m, a, c));
    // Способ 2: присвоить результат символической константе,
    // объявленной с ключевым словом enum
    enum proper1 = properLinearCongruentialParameters(m, a, c);
    // Способ 3: присвоить результат статическому значению
    static proper2 = properLinearCongruentialParameters(m, a, c);
}
```

Мы еще не рассматривали структуры и классы в подробностях, но отметим, немного опережая события, что типичный вариант использования функции properLinearCongruentialParameters – ее размещение внутри структуры или класса, определяющего линейный конгруэнтный генератор. Например:

```
struct LinearCongruentialEngine(UIntType,
    UIntType a, UIntType c, UIntType m) {
```

[1] Распространенный в США и Канаде мем, изначально связанный с фаст-фудом. – *Прим. ред.*

```
        static assert(properLinearCongruentialParameters(m, a, c),
            "Некорректная инициализация LinearCongruentialEngine");

    }
```

Собственно, эти строки скопированы из одноименной структуры, которую можно найти в стандартном модуле `std.random`.

Изменив время выполнения проверки (теперь она выполняется на этапе компиляции, а не во время исполнения программы), мы получили два любопытных последствия. Во-первых, можно было бы отложить проверку до исполнения программы, расположив вызов `properLinearCongruentialParameters` в конструкторе структуры `LinearCongruentialEngine`. Но обычно чем раньше узнаешь об ошибках, тем лучше, особенно если это касается библиотеки, которая почти не контролирует то, как ее используют. При статической проверке некорректно созданные экземпляры `LinearCongruentialEngine` не сигнализируют об ошибках: исключается сама возможность их появления. Во-вторых, используя константы, известные во время компиляции, код имеет хороший шанс работать быстрее, чем код с обычными значениями `m`, `a` и `c`. На большинстве современных процессоров константы в виде литералов могут быть сделаны частью потока команд, так что их загрузка вообще не требует никаких дополнительных обращений к памяти. И посмотрим правде в глаза: линейные конгруэнтные генераторы – не самые случайные в мире, и используют их главным образом благодаря скорости.

Процесс интерпретации на пару порядков медленнее генерации кода, но гораздо быстрее традиционного метапрограммирования на основе шаблонов C++. Кроме того, вычисления во время компиляции (в разумных пределах) в некотором смысле «бесплатны».

На момент написания этой книги у интерпретатора есть ряд ограничений[1]. Выделение памяти под объекты, да и просто выделение памяти запрещены (хотя встроенные массивы работают). Статические данные, вставки на ассемблере и небезопасные средства, такие как объединения (`union`) и некоторые приведения типов (`cast`), также под запретом. Множество ограничений на то, что можно сделать во время компиляции, находится под постоянным давлением. Задумка в том, чтобы разрешить интерпретировать во время компиляции все, что находится в безопасном множестве D. В конце концов, способность интерпретировать код во время компиляции – это новшество, открывающее очень интересные возможности, которые заслуживают дальнейшего исследования.

[1] Многие из этих ограничений уже сняты. – *Прим. науч. ред.*

6

Классы. Объектно-ориентированный стиль

С годами объектно-ориентированное программирование (ООП) из симпатичного малыша выросло в несносного прыщавого подростка, но в конце концов повзрослело и превратилось в нынешнего уравновешенного индивида. Сегодня мы гораздо лучше осознаем не только мощь, но и неизбежные ограничения объектно-ориентированной технологии. В свою очередь, это позволило сообществу программистов понять, что наиболее выгодный подход к созданию надежных проектов – сочетать сильные стороны ООП и других парадигм программирования. Это довольно отчетливая тенденция: все больше современных языков программирования или включают эклектичные средства, или изначально разработаны для применения ООП в сочетании с другими парадигмами. D принадлежит к последним, и его достижения в сфере гармоничного объединения разных парадигм программирования некоторые даже считают выдающимися. В этой главе исследуются объектно-ориентированные средства D и их взаимодействие с другими средствами языка. Хорошая стартовая площадка для глубокого изучения объектно-ориентированной парадигмы – классический труд Бертрана Мейера «Объектно-ориентированное конструирование программных систем» [40] (для более формального изучения лучше подойдут «Типы в языках программирования» Пирса [46, глава 18]).

6.1. Классы

Единицей объектной инкапсуляции в D служит класс. С помощью классов можно создавать объекты, как вырезают печенье с помощью формочек. Класс может определять константы, состояния классов, состояния объектов и методы. Например:

```
class Widget {
   // Константа
   enum fudgeFactor = 0.2;
   // Разделяемое неизменяемое значение
   static immutable defaultName = "A Widget";
   // Некоторое состояние, определенное для всех экземпляров класса Widget
   string name = defaultName;
   uint width, height;
   // Статический метод
   static double howFudgy() {
      return fudgeFactor;
   }
   // Метод
   void changeName(string another) {
      name = another;
   }
   // Метод, который нельзя переопределить
   final void quadrupleSize() {
      width *= 2;
      height *= 2;
   }
}
```

Объект типа `Widget` создается с помощью выражения `new`, результат вычисления которого сохраняется в именованном объекте: `new Widget` (см. раздел 2.3.6.1). Для обращения к идентификатору, определенному внутри класса `Widget`, расположите его после имени объекта, с которым вы хотите работать, и разделите эти два идентификатора точкой. Если член класса, к которому нужно обратиться, является статическим, перед его идентификатором достаточно указать имя класса. Например:

```
unittest {
   // Обратиться к статическому методу класса Widget
   assert(Widget.howFudgy() == 0.2);
   // Создать экземпляр класса Widget
   auto w = new Widget;
   // Поиграть с объектом типа Widget
   assert(w.name == w.defaultName); // Или Widget.defaultName
   w.changeName("Мой виджет");
   assert(w.name == "Мой виджет");
}
```

Обратите внимание на небольшую хитрость. В приведенном коде использовано выражение `w.defaultName`, а не `Widget.defaultName`. Для обращения к статическому члену класса всегда можно вместо имени класса использовать имя экземпляра класса. Это возможно, потому что при обработке выражения слева от точки сначала выполняется разрешение имени и только потом идентификация объекта (если потребуется). Выражение `w` в любом случае вычисляется: будет оно использовано или нет.

6.2. Имена объектов – это ссылки

Проведем небольшой эксперимент:

```
import std.stdio;

class A {
   int x = 42;
}

unittest {
   auto a1 = new A;
   assert(a1.x == 42);
   auto a2 = a1;
   a2.x = 100;
   assert(a1.x == 100);
}
```

Этот эксперимент завершается успешно (все проверки пройдены), а значит, a1 и a2 не являются разными объектами: изменение объекта a2 действительно отразилось и на ранее созданном объекте a1. Эти две переменные – всего лишь два разных имени одного и того же объекта, следовательно, изменение a2 влияет на a1. Инструкция auto a2 = a1; не создает новый объект типа A, а только дает существующему объекту еще одно имя (рис. 6.1).

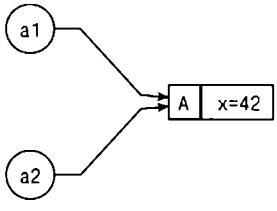

***Рис. 6.1.** Инструкция auto a2 = a1 только вводит дополнительное имя для того же внутреннего объекта*

Такое поведение соответствует принципу: все экземпляры класса являются *сущностями*, то есть обладают «индивидуальностью» и не предполагают копирования без серьезных причин. Экземпляры значения (например, встроенные числа), напротив, характеризуются полным копированием; новый тип-значение определяется с помощью структуры (см. главу 7).

Итак, в мире классов сначала нам встречаются *объекты* (*экземпляры класса*), а затем *ссылки* на них. Воображаемые стрелки, присоединяющие ссылки к объектам, называются *привязками* (*bindings*); мы, например, говорим, что идентификаторы a1 и a2 привязаны к одному и тому же объекту, другими словами, имеют одну и ту же привязку. С объектами

можно работать только через ссылки на них. Получив при создании место в памяти, объект остается там навсегда (по крайней мере до тех пор, пока он вам нужен). Если вам надоест какой-то объект, просто привяжите его ссылку к другому объекту. Например, если нужно, чтобы две ссылки обменялись привязками:

```
unittest {
   auto a1 = new A;
   auto a2 = new A;
   a1.x = 100;
   a2.x = 200;
   // Заставим a1 и a2 обменяться привязками
   auto t = a1;
   a1 = a2;
   a2 = t;
   assert(a1.x == 200);
   assert(a2.x == 100);
}
```

Вместо трех последних строк можно было бы использовать универсальную вспомогательную функцию swap из модуля std.algorithm: swap(a1, a2), но явная запись процесса обмена нагляднее. На рис. 6.2 продемонстрированы привязки до и после обмена.

Сами объекты остаются на том же месте, то есть после создания они никогда не перемещаются в памяти. Просто замечательно, объект никогда не исчезнет: можно рассчитывать, что объект навсегда останется там, куда он был помещен при создании. (Сборщик мусора перерабатывает в фоновом режиме те объекты, которые больше не используются.) Ссылки на объекты (в данном случае a1 и a2) можно заставить «смотреть в другую сторону», переназначив их привязку. Когда библиотека времени исполнения обнаруживает, что для какого-то объекта больше нет привязанных к нему ссылок, она может заново использовать выделенную под него память (этот процесс называется сбором мусора).[1] Такое поведение

[1] Язык D также предоставляет возможность «ручного» управления памятью (manual memory management) и на данный момент позволяет принудительно уничтожать объекты с помощью оператора delete: delete obj;, при этом значение ссылки obj будет установлено в null (см. ниже), а память, выделенная под объект, будет освобождена. Если obj уже содержит null, ничего не произойдет. Однако следует соблюдать осторожность: повторное уничтожение одного объекта или обращение к удаленному объекту по другой ссылке приведет к катастрофическим последствиям (сбои и порча данных в памяти, источники которых порой очень трудно обнаружить), и эта опасность усугубляет необходимость в сборщике мусора. Из-за этих рисков оператор delete планируют убрать из самого языка, оставив в виде функции в стандартной библиотеке. Но при этом ручное управление памятью позволяет более эффективно ее использовать. Вердикт: задействуйте эту возможность, если уверены, что на момент вызова delete объект obj точно не удален и obj – последняя ссылка на данный объект, и не удивляйтесь, если в один прекрасный день delete исчезнет из реализаций языка. – *Прим. науч. ред.*

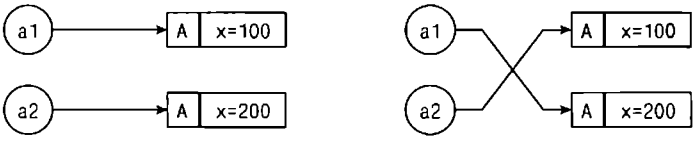

Рис. 6.2. Привязки до и после обмена. В процессе обмена меняются привязки к ссылкам; сами объекты остаются на том же месте

в корне отличается от семантики *значения* (например, `int`), в случае которого нет никаких косвенных изменений или привязок: каждое имя прочно закреплено за значением, которым манипулируют с помощью этого идентификатора.

Ссылка, не привязанная к какому-либо объекту, – это «пустая» ссылка (`null`). При инициализации по умолчанию с помощью свойства `.init` ссылки на классы получают значение `null`. Можно сравнивать ссылку с константой `null` и присваивать ссылке значение `null`. Следующие проверки пройдут успешно:

```
unittest {
   A a;
   assert(a is null);
   a = new A;
   assert(a !is null);
   a = null;
   assert(a is null);
   a = A.init;
   assert(a is null);
}
```

Обращение к элементу непривязанной («пустой», `null`) ссылки ведет к аппаратной ошибке, экстренно останавливающей приложение (или на некоторых системах и при некоторых обстоятельствах запускающей отладчик). Если вы попытаетесь осуществить доступ к нестатическому элементу ссылки и компилятор может статически доказать, что эта ссылка в любом случае в этот момент окажется пустой, он откажется компилировать код.

```
A a;
a.x = 5; // Ошибка! Ссылка a пуста!
```

Иногда компилятор ведет себя сдержанно, стараясь не слишком надоедать вам: если ссылка только *может* быть пустой (но не всегда будет таковой), коду дается «зеленый свет» и все разговоры об ошибках откладываются до времени исполнения программы. Например:

```
A a;
if (‹условие›) {
   a = new A;
}
```

```
if (‹условие›) {
   a.x = 43; // Все в порядке
}
```

Компилятор «пропускает» такой код, даже несмотря на то, что между двумя вычислениями *‹условие›* может изменить значение. В общем случае было бы непросто проверить, насколько корректна инициализация объекта, так что компилятор решает, что вы сами знаете, что делаете (кроме самых простых случаев, когда он уверен, что вы пытаетесь использовать пустую ссылку неподобающим образом).

В языке D применяется такой же основанный на ссылочной семантике подход к обработке объектов классов, как и во многих других объектно-ориентированных языках. Использование для объектов классов ссылочной семантики и сбора мусора имеют как положительные, так и отрицательные следствия, включая следующие:

+ **Полиморфизм.** Уровень косвенности, достигаемый благодаря последовательному использованию ссылок, делает возможной поддержку полиморфизма. Все ссылки обладают одинаковым размером, а ассоциированные с ними объекты могут иметь разные размеры, даже если имеют якобы один и тот же тип (что осуществляется через наследование, о котором речь пойдет очень скоро). Поскольку ссылки обладают одним и тем же размером независимо от размера объектов, на которые они ссылаются, вы всегда можете использовать вместо ссылок на объекты классов-потомков ссылки на объекты родительских классов. Кроме того, как следует работают массивы объектов – даже когда объекты в массиве обладают разными размерами. Если вы имели дело с C++, вам, конечно же, известно о необходимости использования указателей для организации полиморфизма и о разнообразных летальных проблемах, с которыми сталкивается программист, если забывает об этом.

+ **Безопасность.** Многие воспринимают сбор мусора только как удобное средство, которое облегчает процесс кодирования, освобождая программиста от обязанности управлять памятью. Возможно, это прозвучит неожиданно, но модель вечной жизни (которая воплощается благодаря сбору мусора) и безопасность памяти прочно связаны. Там, где жизнь вечна, нет «висячих» ссылок, то есть ссылок на некоторый переставший существовать объект, память которого была заново использована – отдана в распоряжение совершенно постороннего объекта. Заметим, что той же степени безопасности можно добиться, везде используя семантику значения (команда `auto a2 = a1` дублирует экземпляр класса `A`, на который ссылается `a1`, и привязывает `a2` к копии). Такой подход, однако, вряд ли интересен, поскольку лишает возможности создавать какие-либо ссылочные структуры данных (такие как списки, графы и вообще любые разделяемые ресурсы).

– **Цена выделения памяти.** В общем случае классы должны располагаться в куче, подлежащей сбору мусора, что обычно медленнее ра-

ботает и съедает больше памяти, чем при размещении в стеке. В последнее время разница сильно уменьшилась, но она все же есть.

- **Связанность идентификаторов, определенных далеко друг от друга.** Основной риск при использовании ссылок – неумеренное порождение псевдонимов. При повсеместном применении ссылочной семантики очень просто получить ссылки на один и тот же объект в разных – и самых неожиданных – местах. Переменные a1 и a2 на рис. 6.1 могут находиться сколь угодно далеко друг от друга, т. к. по логике приложения кроме них у того же объекта может быть множество других, висячих ссылок. Любопытно, но если объект неизменяем, проблема исчезает: пока никто не изменяет объект, нет и связанности. Сложности возникают, когда некоторое изменение, имевшее место в некотором контексте, неожиданно и драматично повлияет на состояние (как это видится из другой части приложения). Один из способов улучшить такое положение дел заключается в постоянном явном дублировании, которое обычно осуществляется с помощью специального метода clone. Минусы этой техники: она зависит от дисциплинированности человека, и такой образ действий может снизить скорость работы приложения, если некоторые его части решат консервативно клонировать объекты из принципа «как бы чего не вышло».

Сравним ссылочную семантику с семантикой значений а-ля int. У семантики значений есть свои преимущества, среди которых выделяется логический вывод: в выражениях всегда можно заменять равные значения друг на друга, при этом результат не изменяется. (А к ссылкам, использующим для изменения состояния объектов вызовы методов, такой подход неприменим.) Другое важное преимущество семантики значений – скорость. Но даже если вы воспользуетесь динамической щедростью полиморфизма, от ссылочной семантики никуда не деться. Некоторые языки пытались предоставить возможность использовать и ту, и другую семантику и заслужили прозвище «нечистых» (в противоположность чисто объектно-ориентированным языкам, использующим ссылочную семантику унифицированно для всех типов). D нечист и очень гордится этим. Во время разработки необходимо принять решение: если вы желаете работать с некоторым типом в рамках объектно-ориентированной парадигмы, следует выбрать тип class; иначе придется использовать тип struct и поступиться всеми удобствами ООП, присущими ссылочной семантике.

6.3. Жизненный цикл объекта

Теперь, когда мы получили общее представление о местонахождении объекта, подробно изучим его жизненный цикл. Объект создается с помощью выражения new:

```
import std.math;

class Test {
```

```
    double a = 0.4;
    double b;
}

unittest {
    // Объект создается с помощью выражения new
    auto t = new Test;
    assert(t.a == 0.4 && isNaN(t.b));
}
```

При вычислении выражения `new Test` конструируется объект типа `Test` с состоянием по умолчанию, то есть экземпляр класса `Test`, каждое из полей которого инициализировано своим значением по умолчанию. Любой тип `T` обладает статически известным значением по умолчанию, обратиться к которому можно через свойство `T.init` (значения свойств `.init` для базовых типов приведены в табл. 2.1). Если вы хотите инициализировать некоторые поля значениями, отличными от соответствующих значений свойства `.init`, укажите при определении этих полей статически известные инициализирующие значения, как показано в предыдущем примере для поля `a`. Выполнение теста модуля при этом не порождает исключений, так как это поле явно инициализируется константой `0.4`, а поле `b` не трогали, а значит, оно неявно инициализируется значением выражения `double.init`, то есть `NaN` («нечисло»).

6.3.1. Конструкторы

Разумеется, в большинстве случаев бывает недостаточно инициализировать поля лишь статически известными значениями. Выполнить при создании объекта некоторый код позволяет специальная функция – *конструктор*. Конструктор – это функция с именем `this` и без объявления возвращаемого типа.

```
class Test {
    double a = 0.4;
    int b;
    this(int b) {
        this.b = b;
    }
}
unittest {
    auto t = new Test(5);
}
```

Если класс определяет хотя бы один конструктор, то неявный конструктор становится недоступным. С классом `Test`, определенным выше, инструкция

```
auto t = new Test;
```

уже не работает. Цель такого запрета – помочь избежать типичной ошибки: разработчик заботливо определяет ряд конструкторов с пара-

метрами, но совершенно забывает о конструкторе по умолчанию. Как обычно в D такую защиту от забывчивости легко обойти: достаточно показать компилятору, что вы обо всем помните:

```
class Test {
   double a = 0.4;
   int b;
   this(int b) {
      this.b = b;
   }
   this() {} // Конструктор по умолчанию,
             // все поля инициализируются неявно
}
```

Внутри метода (кроме статических методов, см. раздел 6.5) ссылка this неявно привязывается к объекту-адресату вызова. Иногда (как в предыдущем примере, иллюстрирующем общепринятое соглашение об именовании внутри конструкторов) эта ссылка может оказаться полезной: если с помощью параметра a предполагается инициализировать член класса, следует назвать параметр так же, как и член класса, а обращение к последнему уточнить явной ссылкой на объект в виде ключевого слова this, избегая таким образом двусмысленности при обращении к параметру и члену класса.

Несмотря на то что можно изменить свойство this.field для любого поля field, нельзя переназначить привязку самой ссылки this, которая всегда воспринимается компилятором как r-значение:

```
class NoGo {
   void fun() {
      // Просто привяжем this к другому объекту
      this = new NoGo; // Ошибка! Нельзя изменять this!
   }
}
```

Обычные правила перегрузки функций (раздел 5.5) применимы и к конструкторам: класс может определять любое количество конструкторов, но каждый из них должен обладать уникальной сигнатурой (отличающейся числом или типом параметров, хотя бы на один параметр).

6.3.2. Делегирование конструкторов

Рассмотрим класс Widget, определяющий два конструктора:

```
class Widget {
   this(uint height) {
      this.width = 1;
      this.height = height;
   }
   this(uint width, uint height) {
      this.width = width;
      this.height = height;
```

```
    }
    uint width, height;

}
```

В этом коде много повторений, что лишь усугубилось бы в случае классов большего размера, но, к счастью, один конструктор может положиться на другой:

```
class Widget {
    this(uint height) {
        this(1, height); // Положиться на другой конструктор
    }
    this(uint width, uint height) {
        this.width = width;
        this.height = height;
    }
    uint width, height;

}
```

Однако с вызовом конструкторов а-ля this(1, h) связан ряд ограничений. Во-первых, такой вызов возможен только из другого конструктора. Во-вторых, если вы решили сделать такой вызов, то обязаны убедить компилятор, что в вашем конструкторе ровно один такой вызов, несмотря ни на что. Например, следующий конструктор некорректен:

```
this(uint h) {
    if (h > 1) {
        this(1, h);
    }
    // Ошибка! При невыполнении условия конструктор будет пропущен
}
```

В этой ситуации компилятор выяснит, что возможны случаи, когда другой конструктор не будет вызван, и интерпретирует это как ошибку. Смысл такого ограничения в четком разграничении двух альтернатив: конструктор или создает и инициализирует объект сам, или делегирует выполнение этой задачи другому конструктору. Варианты, когда неясно, как поступит конструктор (решит действовать самостоятельно или переложит работу на другого), бракуются.

Дважды вызывать один и тот же конструктор также некорректно:

```
this(uint h) {
    if (h > 1) {
        this(1, h);
    }
    this(0, h);
}
```

Дважды инициализировать объект еще хуже, чем забыть его инициализировать, так что в этом случае также диагностируется ошибка. В двух

словах, конструктору разрешается вызвать другой конструктор или ровно ноль, или ровно один раз. Соблюдение этого правила проверяется во время компиляции с помощью простого анализа потока управления.

6.3.3. Алгоритм построения объекта

Во всех языках построение объекта – не очень простой процесс. Все начинается с получения участка нетипизированной памяти, которая по мере наполнения информацией начинает выглядеть и вести себя как экземпляр класса. И тут никак не обойтись без магии.

В языке D построение объекта включает следующие шаги:

1. *Выделение памяти.* Библиотека времени исполнения выделяет участок «сырой» памяти в куче, достаточный для размещения нестатических полей объекта. Память подо все объекты, основанные на классах, выделяется динамически – в отличие от C++, в D нет способа выделить для объекта память в стеке[1]. Если выделить память не удалось, построение объекта прерывается: порождается исключительная ситуация.

Инициализация полей. Каждое поле инициализируется своим значением по умолчанию. Как уже говорилось, в качестве значения поля по умолчанию выступает значение, указанное при объявлении поля в виде = значение, или при отсутствии такой записи значение свойства .init типа поля.

2. *Брендирование.* После завершения инициализации полей значениями по умолчанию объекту присваивается статус полноправного экземпляра класса T (объект брендируется) еще *до* того, как будет вызван настоящий конструктор.
3. *Вызов конструктора.* Наконец, компилятор инициирует вызов подходящего конструктора. Если класс не определяет собственный конструктор, этот шаг пропускается.

Поскольку объект считается «живым» и правильно построенным сразу после инициализации по умолчанию, настоятельно рекомендуется использовать инициализирующие значения, которые всегда приводят объект в осмысленное состояние. Настоящий конструктор внесет затем свои поправки, приведя объект в другое интересное состояние (разумеется, также осмысленное).

Если ваш конструктор заново присваивает значения некоторым полям, то двойное присваивание не должно быть причиной проблем с быстродействием. В большинстве случаев, если тело конструктора достаточно

---

[1] На данный момент реализации D предоставляют средства выделения памяти под классы в стеке (с помощью класса памяти scope) или вообще в любом фрагменте памяти (с помощью классовых аллокаторов и деаллокаторов). Но поскольку эти возможности небезопасны, они могут быть удалены из языка, так что не рассчитывайте на их вечное существование. – *Прим. науч. ред.*

простое, компилятору хватает ума понять, что первое присваивание лишнее, и применить механизм оптимизации с мрачным названием «уничтожение мертвых присваиваний» (dead assignment elimination).

Если важнее всего эффективность, то в качестве инициализирующего значения поля можно указать ключевое слово void; в этом случае нужно очень внимательно проследить за местонахождением инициализирующего присваивания: оно должно быть внутри конструктора[1]. Возможно, вам покажется удобным использовать = void с массивами фиксированной длины. Оптимизация двойной инициализации всех элементов массива – очень сложная задача для компилятора, и вы можете ему помочь. Следующий код эффективно инициализирует массив фиксированного размера значениями 0.0, 0.1, 0.2, ..., 12.7.

```
class Transmogrifier {
   double[128] alpha = void;
   this() {
      foreach (i, ref e; alpha) {
         e = i * 0.1;
      }
   }

}
```

Иногда некоторые поля намеренно оставляют неинициализированными. Например, экземпляр класса Transmogrifier может отслеживать уже задействованную часть массива alpha с помощью переменной usedAlpha, изначально равной нулю. Таким образом, составные части объекта будут знать, что на самом деле инициализирована только часть массива, а именно элементы с индексами от 0 до usedAlpha - 1:

```
class Transmogrifier {
   double[128] alpha = void;
   size_t usedAlpha;
   this() {
      // Оставить переменную usedAlpha равной 0,
      // а массив alpha - неинициализированным
   }
   .
}
```

Изначально переменная usedAlpha равна нулю, этого достаточно для инициализации объекта класса Transmogrifier. По мере роста usedAlpha код не должен читать элементы в интервале alpha[usedAlpha .. $], а только присваивать им значения. Разумеется, за этим должны следить вы, а не компилятор (вот пример того, что порой эффективность неизбежно связана с проверяемостью на этапе компиляции). Хотя такая оптимизация

[1] В текущих реализациях использование void не влияет на производительность, так как все поля класса инициализируются «одним махом» копированием памяти из .init для экземпляров класса. – *Прим. науч. ред.*

обычно незначительна, иногда лишние принудительные инициализации могут существенно отразиться на общих результатах, и полезно иметь запасной выход.

6.3.4. Уничтожение объекта и освобождение памяти

Для всех объектов классов D поддерживает кучу, пополняемую благодаря сбору мусора. Можно считать, что сразу же после выделения памяти под объект он становится вечным (в пределах времени работы самого приложения). Сборщик мусора перерабатывает память, используемую объектом, только если убежден, что больше нет доступных ссылок на этот объект. Такой подход способствует созданию чистого и безопасного кода, основанного на классах.

Для некоторых классов важно иметь зацепку в процессе уничтожения, чтобы освобождать возможно задействованные ими дополнительные ресурсы. Такие классы могут определять *деструктор*, задаваемый как специальная функция с именем `~this`:

```
import core.stdc.stdlib;

class Buffer {
   private void* data;
   // Конструктор
   this() {
      data = malloc(1024);
   }
   // Деструктор
   ~this() {
      free(data);
   }

}
```

Этот пример иллюстрирует экстремальную ситуацию – класс, который самостоятельно обслуживает собственный буфер «сырой» памяти. В большинстве случаев класс использует должным образом инкапсулированные ресурсы, так что определять деструкторы вовсе не обязательно.

6.3.5. Алгоритм уничтожения объекта

Уничтожение объекта, как и его построение, происходит по определенному алгоритму:

1. Сразу же после брендирования (шаг 3 построения) объект считается «живым» и попадает под наблюдение сборщика мусора. Обратите внимание: это происходит, даже если позже при выполнении определенного пользователем конструктора возникнет исключение. Учитывая, что инициализация по умолчанию и брендирование всегда выполняются успешно, можно сделать вывод, что объект, которому

успешно выделена память, с точки зрения сборщика мусора выглядит как полноценный объект.

2. Объект используется в любом месте программы.
3. Все доступные ссылки на объект исчезли; объект больше недоступен никакому коду.
4. В некоторый момент (зависит от реализации) система осознает, что память объекта может быть переработана, и вызывает деструктор.
5. Спустя еще некоторое время (сразу после вызова деструктора или когда-нибудь позже) система заново использует память объекта.

Важная деталь, имеющая отношение к двум последним шагам: сборщик мусора гарантирует, что деструктор объекта никогда не сможет обратиться к объекту, память которого уже освобождена. Можно обратиться к только что уничтоженному объекту, но не объекту, память которого только что освобождена. В языке D уничтоженные объекты «удерживают» память в течение небольшого промежутка времени – пока не уничтожены связанные с ними объекты. Иначе была бы невозможна безопасная реализация уничтожения и освобождения памяти объектов, ссылающихся друг на друга по кругу (например, кольцевого списка).

Описанный жизненный цикл может варьироваться по разным причинам. Во-первых, очень вероятно, что выполнение приложения завершится еще до достижения даже шага 4, как бывает с маленькими приложениями в системах с достаточным количеством памяти. В таком случае D предполагает, что завершающееся приложение *де-факто* освободит все ресурсы, ассоциированные с ним, так что никакого деструктора язык не вызывает.

Другой способ вмешаться в естественный жизненный цикл объекта – явный вызов деструктора, например с помощью функции `clear` из модуля `object` (модуль стандартной библиотеки, автоматически подключаемый во время каждой компиляции).

```
unittest {
   auto b = new Buffer;

   clear(b); // Избавиться от дополнительного состояния b
   ..  // Здесь все еще можно использовать b
}
```

Вызовом `clear(b)` пользователь выражает желание явно вызвать деструктор `b` (если таковой имеется), стереть состояние этого объекта до `Buffer.init` и установить указатель на таблицу виртуальных функций в `null`, после чего при попытке вызвать метод этого объекта будет сгенерирована ошибка времени исполнения. Тем не менее, в отличие от аналогичной функции в C++, функция `clear` не освобождает память объекта, а в D отсутствует оператор `delete`. (Раньше в D был оператор `delete`, но он уже не используется и считается устаревшим.) Вы все равно можете освободить память, вызвав функцию `GC.free()` из модуля `core.memory`,

если действительно, *действительно* знаете, что делаете. В отличие от освобождения памяти, вызывать функцию clear безопасно, поскольку в этом случае все данные остаются на месте и нет угрозы появления «висячих» указателей. После выполнения инструкции clear(obj); можно, как и раньше, обращаться к объекту obj и использовать его для любых целей, даже если он уже не обладает никаким особенным состоянием. Например, следующий код D считает корректным:

```
unittest {
   auto b = new Buffer;
   auto b1 = b; // Дополнительный псевдоним для b
   clear(b);
   assert(b1.data !is null); // Дополнительный псевдоним все еще ссылается
                             // на (корректный) "скелет" b[1]
}
```

Таким образом, после вызова функции clear объект остается «живым» и пригодным для использования, но его деструктор уже вызван, а состояние объекта соответствует состоянию, которое экземпляры этого класса приобретают после вызова конструктора. Любопытно, что в процессе следующего сбора мусора деструктор объекта будет вызван *снова*. Это происходит, потому что сборщик мусора, естественно, и понятия не имеет о том, в каком состоянии вы оставили объект.

Почему была выбрана такая модель поведения? Ответ прост: благодаря разделению уничтожения объекта и освобождения памяти вы получаете возможность вручную контролировать «дорогие» ресурсы, которые могут находиться в ведении объекта (такие как файлы, сокеты, мьютексы и системные дескрипторы), и одновременно гарантии безопасности памяти. Использование new и clear предохраняет вас от создания висячих указателей. (Встреча с которыми реально угрожает тому, кто якшается с функциями malloc и free из C или с той же функцией GC.free.) В общем случае имеет смысл разделять освобождение ресурсов (безопасно) и переработку памяти (небезопасно). Память в корне отличается от всех других ресурсов, поскольку она представляет собой физическую основу для системы типов. Случайно перераспределив ее, вы рискуете подорвать любые гарантии, которые только может дать система типов.

6.3.6. Стратегия освобождения памяти[2]

На всем протяжении данной главы предлагается, иногда весьма навязчиво, одна стратегия освобождения памяти – использование сборщика мусора. Эта стратегия необычайно удобна, но имеет серьезный недоста-

[1] В соответствии с предыдущим примечанием сегодня этот тест не пройдет. – *Прим. науч. ред.*

[2] Описание этой части языка намеренно не было включено в оригинал книги, но поскольку эта возможность имеется в языке, мы включили ее описание в перевод. – *Прим. науч. ред.*

ток – нерациональное использование памяти. Рассмотрим работу сборщика мусора.

Когда сборщик мусора выделяет некоторую область памяти, он запоминает указатель на эту область и ее размер и начинает вести учет ссылок на данную область. Когда не остается ни одной ссылки на эту область, эта область становится вакантной для освобождения. Но освобождение происходит не сразу.

Если в какой-то момент для выделения очередного блока памяти сборщику мусора не хватает собственного пула памяти, он запускает процедуру сбора мусора, надеясь освободить достаточно памяти для выделения блока запрошенного размера. Если после сбора мусора памяти пула по-прежнему не хватает, сборщик мусора запрашивает память у операционной системы. Процесс сбора мусора сопровождается приостановкой всех потоков выполнения и занимает сравнительно продолжительное время. Впрочем, можно инициировать внеочередной сбор мусора, вызвав функцию `core.memory.GC.collect()`.

Функция `core.memory.GC.disable()` запрещает автоматический вызов процесса сбора мусора, `core.memory.GC.enable()` разрешает его. Если функция `disable` была вызвана несколько раз, то функция `enable` должна быть вызвана как минимум столько же раз для разрешения автоматического сбора мусора.

Функция `core.memory.GC.minimize()` возвращает лишнюю память операционной системе.

Может возникнуть соблазн уничтожать объекты вручную, оставляя сборщику мусора наиболее сложные ситуации, когда отследить жизнь объекта проблематично.

Рассмотрим пример:

```
import std.c.stdlib;
class Storage
{
   private SomeClass sub_obj_1;
   private SomeClass sub_obj_2;
   private void* data;
   this()
   {
      sub_obj_1 = new SomeClass;
      sub_obj_2 = new SomeClass;
      data = malloc(4096);
   }
   ~this()
   {
      free(data);
     // Блок data выделен функцией malloc и не учтен сборщиком мусора,
     // а значит, должен быть уничтожен вручную.
   }
}
```

```
Storage obj = new Storage; // Создали объект Storage

delete obj; // Уничтожили объект
```

Объект obj действительно уничтожен. Что же стало с внутренними объектами obj? Они остались целы, и сборщик мусора их рано или поздно уничтожит. Но предположим, мы уверены, что кроме как в объекте obj ссылок на эти объекты нет, и логично уничтожить их вместе с obj.

Что ж, изменим соответствующим образом деструктор:

```
~this()
{
   delete sub_obj_1;
   delete sub_obj_2;
   free(data);
}
```

Приведенный пример теперь отработает как надо. Но что если объект obj не уничтожать вручную, а оставить на откуп сборщику мусора? В этой ситуации никто не гарантирует, что деструктор obj будет запущен до уничтожения sub_obj_1. Если все произойдет наоборот, то деструктор obj попытается уничтожить уничтоженный объект, что вызовет аварийное завершение программы.

Значит, вызывать деструктирующие функции в деструкторе следует только в том случае, если этот деструктор вызывается вручную, а не сборщиком мусора.

Как определить, кто вызывает деструктор? Для этого достаточно определить собственный деаллокатор (см. раздел 6.15) (если, конечно, ваша реализация компилятора предоставляет такую возможность).

```
class Foo
{
   char[] arr;
   void* buffer;
   this()
   {
      arr = new char[500];
      buffer = std.c.malloc(500);
   }
   ~this()
   {
      std.c.free(buffer);
   }

   private void dispose()
   {
      delete arr;
   }
```

```
    delete(void* v)
    {
        (cast(F)v).dispose();
        core.memory.GC.free(v);
    }
}
```

В данном случае мы переопределяем поведение оператора delete, сообщая ему, что перед непосредственным освобождением памяти следует вызвать метод dispose. В случае уничтожения объекта сборщиком мусора метод dispose вызван не будет.

6.3.7. Статические конструкторы и деструкторы

Внутри классов, как и повсюду в D, статические данные (объявляемые с ключевым словом static) должны всегда инициализироваться константами, известными во время компиляции. Чтобы предоставить легальное средство выполнения кода во время запуска потока, компилятор позволяет определять специальную функцию static this(). Код инициализации на уровне модуля и на уровне класса объединяется, и библиотека поддержки времени исполнения обрабатывает статическую инициализацию в заданном порядке.

Внутри класса можно определить один или несколько статических конструкторов по умолчанию:

```
class A {
    static A a1;
    static this() {
        a1 = new A;
    }

    static this() {
        a2 = new A;
    }
    static A a2;
}
```

Такая функция называется *статическим конструктором класса*. Во время загрузки приложения перед выполнением функции main (а в многопоточном приложении – в момент создания нового потока) runtime-библиотека последовательно выполняет все статические конструкторы в том порядке, в каком они объявлены в исходном коде. В предыдущем примере поле a1 будет инициализировано раньше, чем поле a2. Порядок выполнения статических конструкторов различных классов внутри одного модуля тоже определяется лексическим порядком. Статические конструкторы в модулях, не имеющих отношения друг к другу, выполняются в произвольном порядке. Наконец, самое интересное: статические конструкторы классов в модулях, взаимно зависящих друг от

друга, упорядочиваются так, чтобы исключить возможность использования класса до выполнения его статического конструктора.

Инициализации упорядочиваются так: допустим, класс A определен в модуле MA, а класс B – в модуле MB. Тогда возможны следующие ситуации:

- только один из классов A и B определяет статический конструктор – здесь не нужно беспокоиться ни о каком упорядочивании;
- ни один из модулей MA и MB не включает другой модуль (MB и MA соответственно) – последовательность инициализации классов не определяется (сработает любой порядок, поскольку модули не зависят друг от друга);
- модуль MA включает модуль MB – статический конструктор B выполняется перед статическим конструктором A;
- модуль MB включает модуль MA – статический конструктор A выполняется перед статическим конструктором B;
- модуль MA включает модуль MB и модуль MB включает модуль MA – диагностируется ошибка «циклическая зависимость», и выполнение прерывается на этапе загрузки программы.

Эта цепь рассуждений на самом деле больше зависит не от классов A и B, а от самих модулей и отношений включения между ними. Подробно эти темы обсуждаются в главе 11.

Если при выполнении статического конструктора возникает исключение, выполнение программы также прерывается. Если же все проходит успешно, классам также дается возможность убрать за собой во время останова потока с помощью деструкторов, которые, как и можно было предположить, выглядят так:

```
class A {
   static A a1;
   static ~this() {
      clear(a1);
   }
   ..
   static A a2;
   static ~this() {
      clear(a2);
   }
}
```

Статические деструкторы запускаются в процессе останова потока. Для каждого модуля они выполняются в порядке, *обратном* порядку их определения. В примере выше деструктор a2 будет вызван до вызова деструктора a1. При участии нескольких модулей порядок вызова статических деструкторов, определенных в этих модулях, соответствует порядку, обратному тому, в котором этим модулям давался шанс вызвать

свои статические конструкторы. «Бесконечная башня из черепах»[1] наоборот.

6.4. Методы и наследование

Мы уже стали экспертами по созданию и уничтожению объектов. Пора посмотреть, как можно их использовать. Взаимодействие с объектом заключается в основном в вызове его методов. (В некоторых языках это называется «отправлением сообщений объекту».) Определение метода напоминает определение обычной функции, единственное отличие в том, что определение метода находится внутри класса. Рассмотрим пример. Допустим, было решено создать приложение «Записная книжка», позволяющее сохранять и просматривать контактную информацию. Единицей записываемой и отображаемой информации в таком приложении служит виртуальная визитная карточка, которую можно реализовать в виде класса `Contact`. Кроме прочего можно определить в нем метод, возвращающий цвет фона отображаемой контактной информации:

```
class Contact {
   string bgColor() {
      return "Серый";
   }
}
unittest {
   auto c = new Contact;
   assert(c.bgColor() == "Серый");
}
```

Самое интересное начнется, когда вы решите, что класс должен стать *наследником* другого класса. Например, какие-то контакты в записной книжке относятся к друзьям, и для них хотелось бы использовать другой цвет фона:

```
class Friend : Contact {
   string currentBgColor = "Светло-зеленый";
   string currentReminder;
   override string bgColor() {
      return currentBgColor;
   }
   string reminder() {

      return currentReminder;
   }
}
```

[1] Образ из книги «Краткая история времени» Стивена Хокинга: Вселенная как плоский мир, стоящий на спине гигантской черепахи, «та – на другой черепахе, та – тоже на черепахе, и так все ниже и ниже». – *Прим. ред.*

Объявленный с помощью записи : Contact (наследование от класса Contact), класс Friend будет содержать все, что есть в классе Contact, плюс собственное дополнительное состояние (поля currentBgColor и currentReminder в примере) и собственные методы (метод reminder в примере).

В таких случаях говорят, что класс Friend — это подкласс класса Contact, а класс Contact — суперкласс класса Friend. Благодаря применяемому механизму работы с подклассами можно использовать экземпляр класса Friend везде, где бы ни ожидался экземпляр класса Contact:

```
unittest {
   Friend f = new Friend;
   Contact c = f; // Подставить экземпляр класса Friend
                  // вместо экземпляра класса Contact
   auto color = c.bgColor(); // Вызвать метод класса Friend
}
```

Если бы занявший место экземпляра класса Contact экземпляр класса Friend вел себя *в точности* так же, как и экземпляр ожидаемого класса, отпали бы все (или почти все) причины использовать класс Friend. Одно из основных средств, предоставляемых объектной технологией, — возможность классам-наследникам переопределять функции классов-предков и таким образом модульно настраивать поведение сущностей среды. Как можно догадаться, переопределение задается с помощью ключевого слова override (класс Friend переопределяет метод bgColor), которое обозначает, что вызов c.bgColor() (где вместо c ожидается объект типа Contact, но на самом деле используется объект типа Friend) всегда инициирует вызов версии метода, предлагаемой классом Friend. Друзья всегда остаются друзьями, даже если компилятор думает, что это обыкновенные контакты.

6.4.1. Терминологический «шведский стол»

Технология ООП постоянно и успешно развивалась как в академическом, так и в практическом направлении. На это были затрачены большие усилия, в том числе изобретено великое множество терминов, которые порой сбивают с толку. Остановимся ненадолго и посмотрим на имеющуюся номенклатуру.

Если класс D является прямым наследником класса B, то D называют *подклассом* B, *дочерним классом* B или классом, *производным от* B. А класс B называют *суперклассом*, *родительским классом* (*родителем*) или *базовым классом* D.

Класс X считается потомком класса B, если и только если X является дочерним классом B или X является потомком дочернего класса B. Это рекурсивное определение означает, что если вы посмотрите на родителя X, а потом на родителя родителя X и так далее, то в тот или иной момент вы встретите B.

В этой книге повсеместно используются понятийные пары *родительский класс/дочерний класс* и *предок/потомок*, поскольку эти слова точнее отражают разницу между прямым и косвенным родством, чем пара терминов *суперкласс/подкласс*.

Странно, но несмотря на то что классы – это типы, подтип – это не то же самое, что и подкласс (а супертип – не то же самое, что и суперкласс). Подтип – это более широкое понятие: тип S является подтипом типа T, если значение типа S можно без какого-либо риска употреблять во всех контекстах, где ожидается значение типа T. Обратите внимание: в этом определении ничего не говорится о наследовании. И на самом деле, наследование – это лишь один из способов реализовать порождение подтипов; в общем случае есть и другие средства (в том числе в D). Отношение между порождением подтипов и наследованием можно охарактеризовать так: подтипами класса C являются потомки класса C плюс сам класс C. Подтип C, отличный от C, – это *собственный подтип* (*proper subtype*).

6.4.2. Наследование – это порождение подтипа. Статический и динамический типы

Рассмотрим пример порождения подтипа наследованием. Как уже говорилось, экземпляр класса-потомка всегда может заменить экземпляр своего предка:

```
class Contact {     }
class Friend : Contact {     }
void fun(Contact c) {     }
unittest {
   auto c = new Contact; // c имеет тип Contact
   fun(c);
   auto f = new Friend;  // f имеет тип Friend
   fun(f);
}
```

Несмотря на то что функция fun ожидает экземпляр класса Contact, передача в качестве аргумента объекта f вполне допустима, так как класс Friend является подклассом (и, следовательно, подтипом) класса Contact.

Применяя механизм порождения подтипов, компилятор нередко отчасти «забывает» об истинном типе объекта. Например:

```
class Contact { string bgColor() { return ""; } }
class Friend : Contact {
   override string bgColor() { return "Светло-зеленый"; }
}

unittest {
   Contact c = new Friend; // c имеет тип Contact,
   // но на самом деле ссылается на экземпляр класса Friend
   assert(c.bgColor() == "Светло-зеленый");
```

```
    // Это действительно друг!
}
```

Учитывая, что переменная c имеет тип Contact, ее можно использовать только так, как может быть использован объект типа Contact, – даже если она привязана к объекту типа Friend. Например, нельзя вызвать c.reminder, поскольку этот метод специфичен для класса Friend и отсутствует в классе Contact. Тем не менее команда assert в примере показывает, что друзья всегда остаются друзьями: вызов c.bgColor доказывает, что вызывается метод класса Friend. Как было описано в разделе 6.3, после завершения построения объекта он становится вечным, так что экземпляр класса Friend, созданный с помощью оператора new, никогда никуда не исчезнет. Любопытная особенность, с которой мы столкнулись, состоит в том, что *ссылка* c, привязанная к нему, имеет тип Contact, а не Friend. В таких случаях говорят, что c обладает статическим типом Contact и динамическим типом Friend. Ни к чему не привязанная ссылка (null) не имеет динамического типа.

Отделить тип Friend от маски типа Contact, за которой он скрывается, – или в общем случае потомка от предка – задача посложнее. Есть одно но: операция может закончиться неудачей. Что если на самом деле контакт не ссылается на экземпляр класса Friend? В большинстве случаев компилятор не сможет сказать, так это или нет. Выполнить такое извлечение поручим оператору cast:

```
unittest {
    auto c = new Contact; // Статический и динамический типы переменной
                          // c совпадают. Это Contact
    auto f = cast(Friend) c;
    assert(f is null);    // Переменная f имеет статический тип Friend
                          // и ни к чему не привязана
    c = new Friend;       // Статический: Contact, динамический: Friend
    f = cast(Friend) c;   // Статический: Friend, динамический: Friend
    assert(f !is null);   // Есть!
}
```

6.4.3. Переопределение – только по желанию

Ключевое слово override – это обязательная часть сигнатуры метода Friend.bgColor. Поначалу это может немного раздражать. В конце концов компилятор мог бы сам понять, что выполняется переопределение, и соответствующим образом все увязать. Зачем обязательно писать override?

Ответ связан с сопровождением кода. Компилятору на самом деле ничего не стоит автоматически выяснить, какие методы вы пожелали переопределить. Проблема в том, что он не может определить, какие методы вы *не* хотели переопределять. Такая ситуация может возникнуть, когда вы решаете изменить класс-предок уже после того, как определили класс-потомок. Представьте, например, что изначально в классе Contact определен только метод bgColor. Вы производите от него класс Friend

и переопределяете метод bgColor, как это показано в предыдущем фрагменте кода. Вы также можете определить в классе Friend и другой метод, например метод Friend.reminder, позволяющий извлекать напоминания о некотором конкретном друге. Если позже кто-то еще (или вы сами спустя три месяца) определит метод reminder для класса Contact с иным смыслом, то получит странную неполадку: вызовы, адресованные методу Contact.reminder, будут перенаправляться методу Friend.reminder независимо от того, кому они адресованы, классу Contact или классу Friend, — класс Friend к этому явно не готов.

Обратная ситуация, по крайней мере, настолько же пагубна. Скажем, мы поменяли свои планы и решили удалить или переименовать один из методов класса Contact. Разработчику придется вручную пройти всех потомков класса Contact и решить, что делать с осиротевшими методами. Такие действия почти всегда приводят к ошибкам, и иногда их просто невозможно выполнить во всей полноте, поскольку разные части иерархии классов могут писать разные разработчики.

Таким образом, требование писать override позволяет модифицировать классы-предки без риска неожиданно навредить классам-потомкам.

6.4.4. Вызов переопределенных методов

Иногда метод, перекрывший другой метод, хочет вызвать как раз именно его. Рассмотрим, например, графический виджет Button, наследник класса Clickable. Класс Clickable знает, как рассылать сообщения о нажатиях кнопки объектам-слушателям, но совершенно не в курсе относительно каких-либо графических эффектов. Чтобы обеспечить визуальную реакцию на клик, класс Button переопределяет метод onClick, определенный в классе Clickable, задает код, реализующий все необходимое для визуального эффекта, и желает вызвать метод Clickable.onClick, который выполнил бы все, что касается рассылки сообщений.

```
class Clickable {
   void onClick() {  .. }
}
class Button : Clickable {
   void drawClicked() {     }
   override void onClick() {
      drawClicked();   // Реализует графический эффект
      super.onClick(); // Рассылает слушателям сообщение о нажатии кнопки
   }
}
```

Вызвать метод, который был переопределен, можно с помощью встроенного псевдонима super, предписывающего компилятору обратиться к ранее определенному методу родительского класса. Таким способом можно вызвать любой метод — необязательно переопределенный в классе, откуда делается вызов (например, в методе Button.onClick можно сделать вызов вида super.onDoubleClick). Если честно, идентификатор, к которому

вы обращаетесь, даже не обязан быть именем метода. С таким же успехом это может быть имя поля и вообще любой другой идентификатор. Например:

```
class Base {
   double number = 5.5;
}
class Derived : Base {
   int number = 10;
   double fun() {
      return number + super.number;
   }
}
```

Метод Derived.fun обращается к собственному полю, а также к полю родительского класса, которое по стечению обстоятельств имеет другой тип.

Формат Имякласса.имячленакласса, служит для обращений ко внутренним элементам не только родительского класса, но и любого предка. На самом деле, ключевое слово super – не что иное, как псевдоним, замещающий имя текущего класса-родителя. В предыдущем примере совершенно безразлично, что написать: Base.number или super.number. Очевидная разница лишь в том, что ключевое слово super помогает создать код, который легче сопровождать: если родительский класс изменится, то вам не потребуется искать обращения к нему и заменять старое имя на новое.

Используя имена классов явно, можно прыгнуть вверх по иерархии наследования больше чем на один уровень. Уточнение имени метода с помощью ключевого слова super или имени класса ускоряет обращение к этому методу, поскольку дает компилятору точную информацию о том, какой функции нужно передать управление. Если адресуемый идентификатор не является именем переопределенного метода, на скорость такое уточнение никак не повлияет, только на область видимости.

Несмотря на то что деструкторы (см. раздел 6.3.4) – это всего лишь методы, у обработки вызова деструктора есть особенности. Нельзя явно вызвать деструктор родителя, однако при вызове деструктора текущего класса (или во время цикла сбора мусора, или в результате вызова clear(obj)) библиотека D поддержки во время выполнения всегда вызывает все деструкторы вверх по иерархии.

6.4.5. Ковариантные возвращаемые типы

Продолжим пример с классами Widget, TextWidget и VisibleWidget. Предположим, вы хотите добавить код, порождающий копию экземпляра класса Widget. Если дублируемый объект является экземпляром класса Widget, копия также должна быть экземпляром класса Widget, если дублируемый объект – экземпляр класса TextWidget, то копия также экземпляр класса TextWidget и т. д. Для корректного копирования можно

определить в родительском классе метод duplicate и потребовать, чтобы каждый класс-потомок тоже реализовал метод с таким именем:

```
class Widget {

   this(Widget source) {
          // Скопировать состояние
   }
   Widget duplicate() {
      return new Widget(this); // Выделяет память
                               // и вызывает this(Widget)
   }
}
```

Пока все идет хорошо. Теперь посмотрим на соответствующее переопределение в классе TextWidget:

```
class TextWidget : Widget {

   this(TextWidget source) {
      super(source);
          // Скопировать состояние
   }
   override Widget duplicate() {
      return new TextWidget(this);
   }
}
```

Все корректно, но заметна потеря статической информации: метод TextWidget.duplicate на самом деле возвращает экземпляр класса Widget, а не экземпляр класса TextWidget. Однако если заглянуть *внутрь* функции TextWidget.duplicate, то можно увидеть, что она возвращает TextWidget. Тем не менее эта информация теряется, как только TextWidget.duplicate возвращает результат, поскольку возвращаемым типом этого метода является тип Widget – тот же, что и у метода Widget.duplicate. Поэтому следующий код не работает (хотя в идеале должен бы работать):

```
void workWith(TextWidget tw) {
   TextWidget clone = tw.duplicate(); // Ошибка!
   // Нельзя преобразовать экземпляр класса Widget
   // в экземпляр класса TextWidget!

}
```

Чтобы максимизировать количество доступной статической информации о типах, D вводит средство, известное как *ковариантные возвращаемые типы*. Звучит довольно громко, но смысл ковариантности возвращаемых типов довольно прост: если родительский класс возвращает некоторый тип C, то переопределенной функции разрешается возвращать не только C, но и любого потомка C. Благодаря этому средству можно позволить методу TextWidget.duplicate возвращать TextWidget. Не менее важно, что теперь вы можете прибавить себе веса в дискуссии,

вставив при случае фразу «ковариантные возвращаемые типы». (Шутка. Если серьезно, даже не пытайтесь.)

6.5. Инкапсуляция на уровне классов с помощью статических членов

Иногда бывает полезно инкапсулировать в классе не только поля и методы, но и обычные функции, и (вот это да!) глобальные данные. Такие функции и данные не имеют какого-либо особого предназначения, кроме создания контекста внутри класса. Чтобы сделать обычные функции и данные разделяемыми между всеми объектами класса, определите их с ключевым словом `static`:

```
class Widget {
   static Color defaultBgColor;
   static Color combineBackgrounds(Widget bottom, Widget top) {
      ...
   }
}
```

Внутри статических методов нельзя использовать ссылку `this`. Это также объясняется тем, что статические методы – всего лишь обычные функции, определенные внутри класса. Логически из этого следует, что для получения доступа к данным `defaultBgColor` или к функции `combineBackgrounds` не нужен никакой объект – достаточно только имени класса:

```
unittest {
   auto w1 = new Widget, w2 = new Widget;
   auto c = Widget.defaultBgColor;
   // Сработает и так: w1.defaultBgColor;
   c = Widget.combineBackgrounds(w1, w2);
   // Сработает и так: w2.combineBackgrounds(w1, w2);
}
```

Также вполне корректно обращаться к статическим внутренним элементам класса по имени объекта, а не класса. Обратите внимание: значение объекта будет вычислено в любом случае, даже когда на самом деле в этом нет необходимости.

```
// Создает экземпляр класса Widget и тут же выбрасывает полученный объект
auto c = (new Widget).defaultBgColor;
```

6.6. Сдерживание расширяемости с помощью финальных методов

Иногда бывает нужно запретить подклассам переопределять некоторый метод. Это обычная практика, поскольку методы определяются не для того, чтобы служить лазейками для внесения изменений. В отдельных случаях требуется поддерживать определенные потоки управления

в четко заданном состоянии. (Вспоминается шаблон проектирования «Шаблонный метод» [27].) Чтобы запретить классам-наследникам переопределять метод, определите его с ключевым словом final.

Рассмотрим пример приложения-тикера, формирующего биржевые сводки. Ему необходимо гарантировать своевременное обновление информации на экране при изменении котировок:

```
class StockTicker {
   final void updatePrice(double last) {
      doUpdatePrice(last);
      refreshDisplay();
   }
   void doUpdatePrice(double last) { .. }
   void refreshDisplay() { .. }
}
```

Методы doUpdatePrice и refreshDisplay переопределяемы, а значит, могут быть изменены подклассами класса StockTicker. Например, некоторые тикеры могут определять триггеры и уведомления, срабатывающие только при заданных изменениях котировок, или отображать себя особым цветом. А вот метод updatePrice переопределить нельзя, поэтому инициатор вызова может быть уверен, что при обновлении котировки она обновится и на экране. На самом деле, как истинные борцы за корректность, мы должны определить метод updatePrice так:

```
final void updatePrice(double last) {
   scope(exit) refreshDisplay();
   doUpdatePrice(last);
}
```

Благодаря конструкции scope(exit) информация на экране обновится корректно, даже если при выполнении метода doUpdatePrice возникнет исключение. Такой подход реально гарантирует, что устройство вывода отображает самое свежее и корректное состояние объекта.

У финальных методов есть опасное преимущество, способное легко увлечь вас на темную сторону преждевременной оптимизации. Истина в том, что финальные методы могут быть эффективнее других, потому что при каждом вызове нефинальных методов делается один косвенный шаг, который гарантирует гибкость, обещанную ключевым словом override. Правда, некоторым финальным методам этот шаг также необходим. Например, если вызов финального переопределенного метода инициируется из класса-родителя, он обычно также подвержен косвенным вызовам; в общем случае компилятор по-прежнему не будет знать, куда пойдет вызов. Но если финальный метод определен впервые (а не переопределяет метод родительского класса), то когда бы вы его ни вызвали, компилятор будет на 100% уверен в том, куда «приземлится» вызов. Так что финальные методы, которые ничего не переопределяют, никогда не подвергаются косвенным вызовам; напротив, они наслаждаются теми же правилами вызова, низкими накладными расходами

и возможностями инлайнинга, что и обычные функции. Может показаться, что финальные непереопределяющие методы гораздо быстрее остальных, но это преимущество нивелируют два фактора.

Во-первых, накладные расходы, связанные с вызовом из родительского класса, оцениваются в контексте функции, которая ничего не делает. Но чтобы оценить накладные расходы, которые что-то значат, кроме накладных расходов на инициирование выполнения нужно учитывать время, затрачиваемое на само выполнение кода внутри функции. Если функция короткая, относительные накладные расходы могут быть значительными, но чем более нетривиальным делом занимается функция, тем быстрее уменьшаются относительные накладные расходы, практически сходя на нет. Во-вторых, есть множество техник оптимизации работы компилятора, сборщика и библиотеки поддержки времени исполнения, эффективно работающих на минимизацию и полное уничтожение накладных расходов диспетчирования. Без сомнений, гораздо удобнее начать с гибкого кода, оптимизируя очень умеренно, а не сразу наделять методы чрезмерной строгостью, ограничивающей их возможности, ради потенциального быстродействия в отдаленном будущем.

Если вы работали с языками Java или C#, то немедленно узнаете в ключевом слове `final` старого знакомого, поскольку в D оно обладает той же семантикой, что и в этих языках. Если сравнить положение дел с C++, то можно обнаружить любопытную смену настроек по умолчанию: в C++ методы являются финальными по умолчанию (и не нужно специально что-то указывать, чтобы запретить наследование) и переопределяемыми, если явно пометить их ключевым словом `virtual`. Подчеркнем еще раз: по крайней мере в этом случае было решено предпочесть умолчания, ориентированные на гибкость. Скорее всего, вы будете использовать финальные методы в основном для реализации структурных решений и только иногда – чтобы избавиться от нескольких дополнительных циклов процессора.

6.6.1. Финальные классы

Иногда требуется, чтобы класс «закрыл тему». Для этого можно пометить ключевым словом `final` целый класс:

```
class Widget {     }
final class UltimateWidget : Widget {     }
class PostUltimateWidget : UltimateWidget {     } // Ошибка!
                // Нельзя стать наследником финального класса
```

От финального класса нельзя наследовать – в иерархии наследования он является листом. Иногда это может оказаться важным средством разработки. Очевидно, что все методы финального класса также неявно объявляются как финальные: их никогда нельзя будет переопределить.

Любопытный побочный эффект финальных классов – твердые гарантии реализации. Клиентский код, использующий финальный класс,

может быть уверен, что методы этого класса обладают известными реализациями с гарантированным действием, которое не может быть модифицировано никаким подклассом.

6.7. Инкапсуляция

Одна из характерных черт объектно-ориентированной, да и других техник разработки – *инкапсуляция*. Объект инкапсулирует детали своей реализации, выставляя напоказ лишь тщательно выверенный интерфейс. Таким способом объекты закрепляют за собой свободу изменять множество деталей своей реализации без угрозы для работоспособности клиентского кода. Это позволяет уменьшить количество связей в коде и, следовательно, количество зависимостей, подтверждая знаменитую истину: каждая техника разработки в конечном счете стремится облегчить управление зависимостями.

В то же время инкапсуляция – это проявление принципа *сокрытия информации* (*information hiding*), одного из основных для разработки программного обеспечения. Этот принцип гласит, что множество логически обособленных частей приложения должны определять и использовать для взаимодействия друг с другом абстрактные интерфейсы, скрывая детали их реализации. Обычно эти детали касаются структур данных, поэтому распространено понятие «сокрытие данных» (data hiding). Тем не менее сокрытие данных – лишь частный случай сокрытия информации, поскольку компонент может скрывать множество разнообразной информации, в том числе структурные решения и алгоритмические стратегии.

Сегодня инкапсуляция кажется благом, практически несомненным, но во многом такое отношение – результат накопленного коллективного опыта. Раньше все было не столь ясно и однозначно. В конце концов информация – вроде бы хорошая вещь, чем ее больше – тем лучше. С какой стати ее прятать?

Отмотаем пленку назад. В 1960-х Фред Брукс, автор основополагающей книги «Мифический человеко-месяц»[1], выступал в поддержку прозрачного («белый ящик») подхода к разработке программного обеспечения под девизом «все знают всё». Под его руководством команда, работающая над операционной системой OS/360, регулярно получала документацию со сведениями обо всех деталях проекта благодаря замысловатой методике, основанной на печати документирующих комментариев [13, глава 7]. Проект был довольно успешным, но вряд ли можно утверждать, что прозрачность сыграла в этом серьезную роль; гораздо более правдоподобно то, что эта прозрачность была риском, минимизированным за счет усиленного управления. Окончательно причислить сокрытие информации к непререкаемым принципам сообщества программистов по-

[1] Ф. Брукс «Мифический человеко-месяц». – Символ-Плюс, 2000.

могло только появление революционного сочинения Дэвида Парнаса [44]. В 1995 году Брукс сам отметил, что его пропаганда прозрачности – единственное в «Мифическом человеко-месяце», что не прошло проверку временем. Но в 1972 году мысль о сокрытии информации вызывала полемику, о чем свидетельствует отзыв рецензента революционного сочинения Парнаса: «Очевидно, что Парнас не знает, о чем говорит, потому что никто так не делает». Довольно забавно, что десяток лет спустя положение дел изменилось настолько радикально, что то же сочинение стало почти банальностью: «Парнас лишь записал то, что и так делали все хорошие программисты» [32, с. 138].

Вернемся к инкапсуляции в контексте языка D. Любые тип, данные, функцию или метод можно определить с одним из следующих пяти спецификаторов. Начнем с самого закрытого спецификатора и постепенно дойдем до полной гласности.

6.7.1. private

Спецификатор доступа `private` можно указывать на уровне класса, за пределами классов (на уровне модуля) или внутри структуры (см. главу 7). Во всех контекстах ключевое слово `private` действует одинаково – ограничивает доступ к идентификатору до текущего модуля (файла).

В других языках такой синтаксис имеет другую семантику: обычно доступ к закрытым идентификаторам ограничивают лишь до текущего класса. Тем не менее то, что спецификатор `private` ограничивает доступ до уровня модуля, прекрасно согласуется с общим подходом D к защите: все защищаемые единицы соответствуют защищаемым единицам операционной системы (файлу и каталогу). Преимущество защиты на уровне файла в том, что она способствует объединению маленьких тесно взаимосвязанных сущностей, обладающих определенными обязанностями. Если требуется защита на уровне класса, просто выделите ему собственный файл.

6.7.2. package

Спецификатор доступа `package` можно указывать на уровне класса, за пределами классов (на уровне модуля) и внутри структуры (см. главу 7). Во всех контекстах ключевое слово `package` действует одинаково: доступ к идентификатору, определенному с этим ключевым словом, предоставляется всем файлам в том же каталоге, где находится и текущий модуль. Родительский каталог и подкаталоги каталога текущего модуля не обладают никакими особыми привилегиями.

6.7.3. protected

Спецификатор доступа `protected` имеет смысл только внутри класса, но не на уровне модуля. При использовании внутри некоторого класса `C` этот спецификатор доступа означает, что доступ к объявленному иден-

тификатору сохраняется за модулем, в котором определен класс C, а также за всеми потомками класса C независимо от того, в каком модуле они находятся. Например:

```
class C {
   // Поле x доступно только в этом файле
   private int x;
   // Этот файл и все прямые и косвенные наследники класса C
   // могут вызвать метод setX()
   protected void setX(int x) { this.x = x; }
   // Кто угодно может вызвать метод getX()
   public int getX() { return x; }
}
```

И снова отметим, что право доступа, предоставляемое спецификатором доступа protected, транзитивно: оно переходит не только собственно к дочерним классам, но и ко всем поколениям потомков, наследующих от класса, определенного с ключевым словом protected. Это делает спецификатор доступа protected довольно щедрым в плане предоставления доступа.

6.7.4. public

Общедоступный уровень доступа public означает, что к объявленному таким способом идентификатору можно свободно обращаться из любого места в приложении. Все, что должно для этого сделать приложение, – поместить нужный идентификатор в свою область видимости (как правило, это делается с помощью импорта модуля, в котором объявлен данный идентификатор).

В языке D public – это также уровень доступа, который присваивается всем объектам по умолчанию. Поскольку порядок объявлений на компиляцию не влияет, хорошим тоном будет расположить видимые интерфейсы модуля или класса в начале файла, а затем ограничить доступ, применив (например) спецификатор доступа private, после чего можно разместить другие определения. Если придерживаться такой стратегии, клиенту будет достаточно посмотреть в начало файла или класса, чтобы узнать все о его доступных сущностях.

6.7.5. export

Казалось бы, спецификатор доступа public – наименее закрытый из уровней доступа, самый щедрый из них. Тем не менее D определяет уровень доступа, разрешающий еще больше: export. Идентификатор, определенный с ключевым словом export, становится доступным даже *вне* программы, в которой он был определен. Это случай разделяемых библиотек, выставляющих всему миру напоказ свои интерфейсы. Компилятор выполняет зависящие от системы шаги, необходимые для экспорта идентификатора, часто включая и особые соглашения об именовании символов. Пока что в D не определена сложная инфраструктура

динамической загрузки, так что спецификатор доступа export выполняет роль заглушки в ожидании более обширной поддержки.

6.7.6. Сколько инкапсуляции?

Логичный и интересный вопрос: «Как сравнить пять определенных в D уровней доступа?» Например, мы только что согласились, что скрывать информацию – хорошо, и резонно было бы считать, что уровень доступа private «лучше», чем protected, поскольку у первого больше ограничений. Далее, по той же логике protected лучше, чем public (еще бы – public довольно низко опускает планку, про export можно и не говорить). При этом неясно, как сравнить protected и package. А главное, такой «качественный» анализ даже не намекает на то, какие потери понесет разработчик, решивший, к примеру, смягчить ограничения для идентификатора. К чему ближе спецификатор доступа protected – к private или public? Или он ровно посередине шкалы? И что это за шкала в конце концов?

Давным-давно, в декабре 1999 года, когда всех волновала лишь проблема 2000 года, Скотта Мейерса волновала инкапсуляция, точнее методы программирования, позволяющие ее максимизировать. Результатом его исследований стала статья [41], в которой Мейерс предложил простой критерий для оценки «степени инкапсуляции» сущности: если мы изменим сущность, сколько кода затронут наши изменения? Чем меньше кода будет затронуто, тем большая степень инкапсуляции достигнута.

Понимание смысла измерения степени инкапсуляции многое проясняет. В отсутствие системы измерения о спецификаторах доступа обычно судят так: «private – хорошо, public – плохо, а protected – нечто среднее между ними». Человек по своей природе оптимист, поэтому уровень защиты, предоставляемый спецификатором доступа protected, многие оценивали как «хороший в известных пределах», а-ля «умеренно пьющий».

Другой аспект, который можно использовать для оценки степени инкапсуляции, – это *контроль*, то есть ваше влияние на код, в который вносятся изменения. Знаете ли вы (или: легко ли вы отыщете) код, на котором отразятся изменения? Обладаете ли вы правами на изменение этого кода? Может ли кто-то еще добавлять в него что-то? Ответы на эти вопросы определяют степень вашего контроля над кодом.

Для начала рассмотрим спецификатор доступа private. Изменение закрытого идентификатора затрагивает ровно один файл. Обычный размер исходного кода – примерно тысяча строк, нередко встречаются файлы и меньшего размера; более объемные файлы (скажем, в 10000 строк) труднее сопровождать. Поскольку вы изменяете только один файл, можно сделать вывод, что вы обладаете над ним полным контролем и легко ограничите доступ к нему для других людей с помощью его атрибутов, системы управления версиями или стандартов кодирования, принятых в команде. Итак, спецификатор доступа private предоставляет блестящую инкапсуляцию: изменения затрагивают небольшое количество кода, при этом степень вашего контроля над кодом достаточно высока.

При использовании спецификатора доступа package изменения затронут все файлы в том же каталоге. Можно прикинуть, что объем содержимого файлов, объединенных в пакет, составит примерно на порядок больше строк (например, разумно считать, что пакет включает примерно десять модулей). Соответственно изменять символы пакета недешево: изменения затронут код на порядок большего размера, чем при аналогичных изменениях private-идентификаторов. К счастью, у вас все еще хороший контроль над кодом, на котором отразятся изменения, поскольку опять же операционная система и разнообразные инструменты управления версиями предоставляют контроль над добавлением и изменением файлов на уровне каталога.

К сожалению, «защищенный» спецификатор доступа protected предоставляет гораздо меньшую защиту, чем обещает его название. Во-первых, со спецификатора protected начинается ощутимое расширение границ доступа, определяемых спецификаторами private и package: любой класс, расположенный где угодно в программе, может получить доступ к защищенному идентификатору, просто создав потомок класса, определяющего этот идентификатор. У вас же из средств «мелкодисперсного» контроля за наследованием – только атрибут final с девизом «всё или ничего». Из этого следует, что изменив защищенный идентификатор, вы повлияете на неограниченное количество кода. Усугубляет ситуацию то, что вы не только не можете ограничить тех, кто наследует от вашего класса, но еще и можете испортить код, на исправление которого у вас нет прав. (Например, изменение идентификатора библиотеки повлияет на все использующие ее приложения.) Реальность становится столь же мрачной, сколь и хрупкой: начав изменять что-то помимо private и package, вы открыты всем ветрам. В «защищенном» режиме доступа protected вы практически беззащитны.

Сколько кода придется просмотреть при изменении защищенного идентификатора? Всех наследников класса, в котором этот идентификатор определен. Разумная приблизительная оценка – на порядок больше размера пакета, то есть несколько сотен тысяч строк. Здесь, конечно, очень помогут инструменты, индексирующие исходный код и отслеживающие наследников классов, но, что ни говори, изменение защищенного символа может затронуть огромные объемы кода.

Со спецификатором доступа public вы не заметите ощутимых перемен в контроле, но зато обнаружите, что затрагиваемый код вырос еще на порядок. Теперь это не только потомки классов, но и весь остальной код приложения. Наконец, спецификатор доступа export добавляет к сюжету еще один интересный поворот: изменяя экспортируемые идентификаторы, вы рискуете повлиять на все бинарные приложения, использующие ваш код как бинарную библиотеку, так что вы не просто смотрите на код, который не можете исправить, – это код, который нельзя даже увидеть, потому что он может быть недоступен в исходном виде.

На рис. 6.3 эти приблизительные оценки изображены в виде графика зависимости числа потенциально затронутых строк кода от каждого спецификатора доступа. Конечно, эти числа лишь ориентировочные, на практике значения могут варьироваться в широких пределах, но основные пропорции вряд ли сильно изменятся. Шкала вертикальной оси – логарифмическая, ступени роста обозначают линейный рост, так что снизив защиту доступа всего на йоту, вы будете работать примерно в десять раз усерднее, чтобы синхронизировать все части кода. Стрелки вверх означают потерю контроля над затронутым кодом. Один из выводов заключается в том, что `protected` находится не ровно посередине между `private` и `public`, а гораздо ближе к `public`, и относиться к нему нужно так же (то есть с животным страхом).

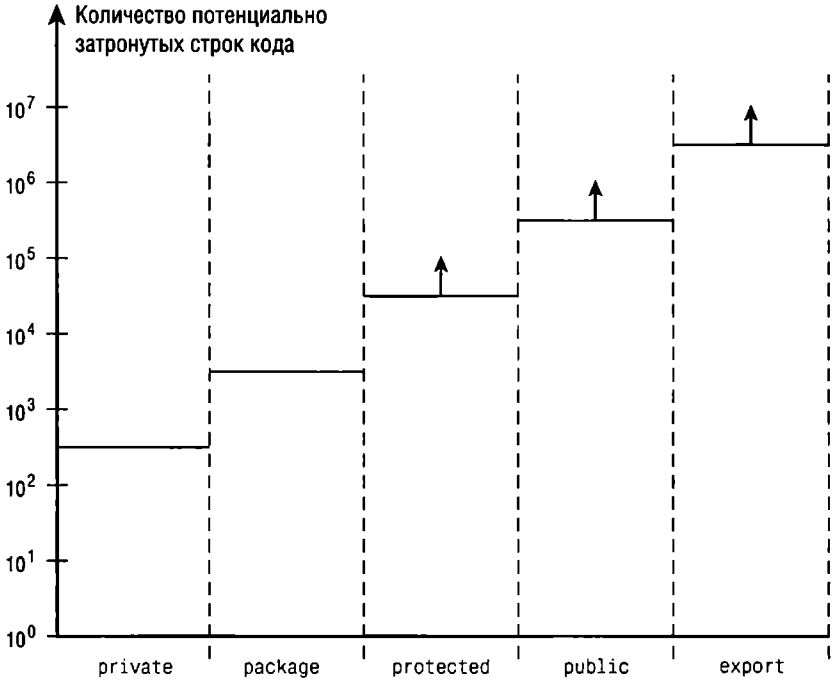

***Рис. 6.3.** Приблизительные оценки количества строк кода, которые может затронуть изменение идентификатора с соответствующим спецификатором доступа. Вертикальная ось – логарифмическая, так что каждый шаг ослабления инкапсуляции на порядок ухудшает положение дел. Стрелки вверх означают, что количество кода, затронутого при уровне защиты `protected`, `public` и `export`, неподвластно программисту, изменившему идентификатор*

6.8. Основа безраздельной власти

В языке D, как и в некоторых других языках, определен корневой класс для всех остальных классов. Всеобщий корень называется `Object`. Когда вы определяете класс, например так:

```
class C {

}
```

компилятор распознает это как:

```
class C : Object {

}
```

Кроме этой автоматической перезаписи класс `Object` ничем не примечателен – он такой же, как все остальные классы. Ваша реализация определяет его в модуле `object.di` или `object.d`, автоматически включаемом в каждый модуль, который вы компилируете. Просмотрев каталог, где находится ваша реализация D, вы легко обнаружите этот модуль и убедитесь, что он содержит корневой объект.

Определение общего прародителя для всех классов имеет ряд положительных следствий. Очевидное благо – то, что класс `Object` может объявлять ряд повсеместно полезных методов. Вот немного упрощенное определение класса `Object`:

```
class Object {
   string toString();
   size_t toHash();
   bool opEquals(Object rhs);
   int opCmp(Object rhs);
   static Object factory(string classname);
}
```

Познакомимся поближе с семантикой каждого из этих идентификаторов.

6.8.1. string toString()

Этот метод возвращает текстовое представление объекта. По умолчанию метод `toString` возвращает имя класса:

```
// Файл test.d
class Widget {}
unittest {
   assert((new Widget).toString() == "test.Widget");
}
```

Обратите внимание: вместе с именем класса возвращено и имя модуля, в котором класс был определен. По умолчанию модуль получает имя файла, в котором он расположен, но это умолчание можно изменить с помощью объявления с ключевым словом `module` (см. раздел 11.8).

6.8.2. size_t toHash()

Этот метод возвращает хеш объекта в виде целого числа без знака (размером в 32 разряда на 32-разрядной машине и в 64 разряда на 64-разрядной). По умолчанию хеш-сумма вычисляется на основе поразрядного представления объекта. Хеш является сжатым, но неточным представлением объекта. Одно из важных требований к функции, вычисляющей хеш-сумму, – *постоянство*: если метод `toHash` дважды вызывается с одной и той же ссылкой и между двумя вызовами объект, к которому привязана эта ссылка, не был изменен, то значения, возвращенные при первом и втором вызове, должны совпадать. Кроме того, хеш-коды двух одинаковых объектов тоже должны быть одинаковыми. А хеш-коды двух различных («неравных») объектов вряд ли будут равны. В следующем разделе подробно определено понятие равенства объектов.

6.8.3. bool opEquals(Object rhs)

Этот метод возвращает `true`, если объект `this` сочтет, что значение `rhs`[1] ему равно. Это намеренно странная формулировка. Эксперимент с аналогичной функцией `equals` из языка Java показал, что если есть наследование, то определить равенство объектов не так просто. Поэтому D подходит к этому вопросу достаточно своеобразно.

Начнем с того, что у нас уже есть одно определение равенства объектов: выражение `a1 is a2` (см. раздел 2.3.4.3), сравнивающее ссылки на объекты классов `a1` и `a2`, истинно тогда и только тогда, когда `a1` и `a2` ссылаются на один и тот же объект (см. рис. 6.1). Это разумное определение равенства объектов, но чересчур строгое, чтобы быть полезным. Обычно требуется, чтобы два физически разных объекта считались равными, если они находятся в одинаковых состояниях. В языке D логическое равенство вычисляется с помощью операторов `==` и `!=`. Вот как они работают.

Допустим, для выражений *‹lhs›* и *‹rhs›* можно записать: *‹lhs›* `==` *‹rhs›*. Тогда если хотя бы одно из них имеет пользовательский тип, компилятор переписывает сравнение в виде `object.opEquals(`*‹lhs›*`, `*‹rhs›*`)`. Аналогично сравнение *‹lhs›* `!=` *‹rhs›* заменяется на `!object.opEquals(`*‹lhs›*`, `*‹rhs›*`)`. Вспомните, чуть выше уже говорилось, что `object` – стандартный модуль, определенный реализацией D и неявно включаемый с помощью инструкции `import` во все модули вашей сборки. Так что сравнения превращаются в вызовы свободной функции, предоставляемой вашей реализацией и расположенной в модуле `object`.

От отношения равенства ожидается подчинение определенным инвариантам, и выражение `object.opEquals(`*‹lhs›*`, `*‹rhs›*`)` проходит долгий путь, доказывая свою корректность. Во-первых, сравнение пустых ссылок (`null`)

---

[1] `rhs` (от right hand side – справа от) – значение, в выражении расположенное справа от оператора. Аналогично `lhs` (от left hand side – слева от) – значение, в выражении расположенное слева от оператора. – *Прим. ред.*

должно возвращать true. Далее, для любых трех непустых ссылок x, y, z должны успешно выполняться следующие проверки:

```
// Ссылка null уникальна; непустая ссылка не может быть равна null
assert(x != null);
// Рефлексивность
assert(x == x);
// Симметричность
assert((x == y) == (y == x));
// Транзитивность
if (x == y && y == z) assert(x == z);
// Отношение с toHash
if (x == y) assert(x.toHash() == y.toHash());
```

Более тонкое требованиее к методу opEquals — *постоянство*: вычисление равенства дважды с одними и теми же ссылками должно возвращать один и тот же результат, при условии что между первым и вторым вызовом opEquals объекты, к которым привязаны данные ссылки, не изменялись.

Типичная реализация object.opEquals сначала исключает некоторые простые или вырожденные случаи, а затем открывает дорогу конкретным версиям opEquals, определенным для типов выражений, участвующих в сравнении. Вот как может выглядеть object.opEquals:

```
// В системном модуле object.d
bool opEquals(Object lhs, Object rhs) {
   // Если это псевдонимы одного и того же объекта
   // или пустые ссылки, то они равны
   if (lhs is rhs) return true;
   // Если только один аргумент равен null, то не равны
   if (lhs is null || rhs is null) return false;
   // Если типы в точности совпадают, то вызываем метод opEquals один раз
   if (typeid(lhs) == typeid(rhs)) return lhs.opEquals(rhs);
   // В общем случае - симметричные вызовы метода opEquals
   return lhs.opEquals(rhs) && rhs.opEquals(lhs);
}
```

Во-первых, если две ссылки ссылаются на один и тот же объект или обе пустые, то, как и можно было ожидать, результат — true (гарантируется рефлексивность). Далее, если установлено, что объекты индивидуальны, и если один из них равен null, сравнение возвращает false (гарантируется исключительность пустой ссылки null). Третья проверка устанавливает, имеют ли объекты один и тот же тип; если да, то системная функция поручает вычислить результат сравнения выражению lhs.opEquals(rhs). И самый интересный момент: двойное вычисление в последней строке. Почему одного вызова недостаточно?

Вспомните изначальное, немного сложное для понимания описание метода opEquals: «возвращает true, если объект this сочтет, что значение rhs ему равно». Это определение заботится лишь о this, но не принимает во внимание мнение, которое может быть у rhs. Для достижения взаимного

согласия необходимо рукопожатие – каждый из двух объектов должен утвердительно ответить на вопрос: «Считаете ли вы, что этот объект вам равен?» Может показаться, что разногласия относительно равенства – это лишь академическая проблема, но они довольно-таки часто возникают там, где на сцену выходит наследование. Впервые об этом заговорил Джошуа Блох (Joshua Bloch) в своей книге «Effective Java» [9], а продолжил тему Тал Коэн (Tal Cohen) в своей статье [17]. Попробуем проследить эту полемическую цепь рассуждений.

Вернемся к примеру с графическими пользовательскими интерфейсами. Пусть требуется определить графический виджет, связанный с окном:

```
class Rectangle {     }
class Window {     }
class Widget {
   private Window parent;
   private Rectangle position;
    .. // Собственные функции класса Widget
}
```

Затем определяем класс TextWidget – тот же Widget, но отображающий какой-то текст.

```
class TextWidget : Widget {
   private string text;
   ...
}
```

Как реализовать метод opEquals для этих двух классов? В случае класса Widget один объект этого класса будет равен другому, если обладает тем же состоянием:

```
// Внутри класса Widget
override bool opEquals(Object rhs) {
   // Второй объект должен быть экземпляром класса Widget
   auto that = cast(Widget) rhs;
   if (!that) return false;
   // Сравнить всё
   return parent == that.parent
      && position == that.position;
}
```

Выражение cast(Widget) пытается вытянуть объект типа Widget из rhs. Если rhs – это пустая ссылка или действительный, динамический тип rhs не Widget и не подкласс Widget, выражение с приведением типов возвращает null.

Класс TextWidget более тонко понимает равенство: правая часть сравнения должна также быть экземпляром класса TextWidget и содержать тот же текст.

```
// Внутри класса TextWidget
override bool opEquals(Object rhs) {
```

```
    // Второй объект должен быть экземпляром TextWidget
    auto that = cast(TextWidget) rhs;
    if (!that) return false;
    // Сравнить все имеющие отношение к делу состояния
    return super.opEquals(that) && text == that.text;
}
```

Рассмотрим сравнение экземпляра `tw` класса `TextWidget` и экземпляра `w` класса `Widget`, у которого те же расположение и родительское окно. С точки зрения `w` между ним и `tw` наблюдается равенство. Однако с точки зрения `tw` ситуация выглядит иначе: равенство отсутствует, поскольку `w` не является экземпляром класса `TextWidget`. Если бы мы приняли вариант, когда `w == tw`, но `tw != w`, то оператор `==` лишился бы симметричности. Чтобы восстановить симметричность, рассмотрим вариант с менее строгим классом `TextWidget`: пусть внутри метода `TextWidget.opEquals` при обнаружении того, что `rhs` является экземпляром класса `Widget`, но не `TextWidget`, планка сравнения опускается до представления о равенстве класса `Widget`. Реализовать этот метод можно так:

```
// Альтернативный метод TextWidget.opEquals - СЛОМАННЫЙ
override bool opEquals(Object rhs) {
    // Второй объект должен быть хотя бы экземпляром класса Widget
    auto that = cast(Widget) rhs;
    if (!that) return false;
    // Они равны как экземпляры класса Widget? Если нет, мы закончили
    if (!super.opEquals(that)) return false;
    // Это экземпляр класса TextWidget?
    auto that2 = cast(TextWidget) rhs;
    // Если нет, сравнение закончено успешно
    if (!that2) return true;
    // Сравнить как экземпляры класса TextWidget
    return text == that2.text;
}
```

К сожалению, стремление класса `TextWidget` быть более сговорчивым до добра не доведет. Теперь проблема в том, что «сломалась» транзитивность сравнения: легко создать два объекта класса `TextWidget` `tw1` и `tw2`, которые будут отличаться друг от друга (из-за разного текста), но в то же время оба окажутся равными простому экземпляру `w` класса `Widget`. Такое положение дел создало бы ситуацию, когда `tw1 == w` и `tw2 == w`, но `tw1 != tw2`.

Итак, в общем случае сравнение должно выполняться дважды: каждый из операндов сравнения должен подтвердить свое равенство другому операнду. Но есть и хорошие новости: свободная функция `object.opEquals(Object, Object)` избегает процедуры «рукопожатия» – при любом совпадении типов участвующих в сравнении объектов, а в других случаях иногда вообще не инициирует дополнительные вызовы.

6.8.4. int opCmp(Object rhs)

Этот метод реализует упорядочивающее трехвариантное сравнение[1], необходимое для использования объектов в качестве ключей ассоциативных массивов. Он возвращает некоторое отрицательное число, если this меньше rhs; некоторое положительное число, если this больше rhs; и 0, если this и rhs считаются неупорядоченными. Так же как и opEquals, метод opCmp редко вызывают явно. В большинстве случаев вы инициируете его выполнение неявно одним из следующих выражений: a < b, a <= b, a > b и a >= b.

Замена производится по той же схеме, что и в случае метода opEquals: в качестве посредника во взаимодействии двух объектов-участников задействуется глобальное определение object.opCmp. Для каждого из операторов <, <=, > и >= компилятор D переписывает выражение a *‹оп›* b в виде object.opCmp(a, b) *‹оп›* 0. Например запись a < b превращается в object.opCmp(a, b) < 0.

Реализовывать метод opCmp необязательно. Реализация по умолчанию Object.opCmp порождает исключение. Если вы на самом деле реализуете его, метод opCmp должен задавать «строгий слабый порядок», то есть этот метод должен удовлетворять следующим инвариантам для непустых ссылок x, y и z:

```
// 1. Рефлексивность
assert(x.opCmp(x) == 0);
// 2. Транзитивность знака
if (x.opCmp(y) < 0 && y.opCmp(z) < 0) assert(x.opCmp(z) < 0);
// 3. Транзитивность равенства нулю
if ((x.opCmp(y) == 0 && y.opCmp(z) == 0) assert(x.opCmp(z) == 0);
```

Эти три правила могут показаться странными, поскольку выражают аксиомы в терминах не очень знакомого понятия трехвариантного сравнения. Если же переписать их в терминах <, получатся знакомые свойства строгого слабого порядка, как они определены в математике:

```
// 1. Отсутствие y < рефлексивности
assert(!(x < x));
// 2. Транзитивность <
if (x < y && y < z) assert(x < z);
// 3. Транзитивность !(x < y) && !(y < x)
if (!(x < y) && !(y < x) && !(y < z) && !(z < y))
   assert(!(x < z) && !(z < x));
```

Третье условие необходимо, чтобы отношение < можно было считать строгим слабым порядком. Без этого условия < называют *частичным порядком*. Возможно, вам хватит и частичного порядка, но только для ограниченного числа случаев; большинство интересных алгоритмов

[1] Интересно, что семантика использования opCmp та же, что и в функциях сравнения памяти и строк в языке C. – *Прим. науч. ред.*

рассчитаны на строгий слабый порядок. Определять частичный порядок гораздо лучше без синтаксического сахара – с помощью собственных именованных функций, отличных от opCmp.

Заметим, что указанные условия определены лишь для <, о других упорядочивающих сравнениях речь не идет, потому что они – лишь синтаксический сахар (x > y – то же самое, что y < x, а x <= y – то же самое, что !(y > x), и т. д.).

Есть свойство, являющееся лишь следствием транзитивности и отсутствия рефлексивности, которое, тем не менее, иногда принимают за аксиому, – антисимметричность: из x < y следует, что !(y < x). Используя метод *доказательства от противного*, легко удостовериться в том, что не существует такой пары значений x и y, что неравенства x < y и y < x будут справедливы одновременно: если бы это было не так, в предыдущей проверке транзитивности можно было бы заменить z на x, получив такую запись:

```
if (x < y && y < x) assert(x < x);
```

Если следовать нашей гипотезе, это сравнение истинно, следовательно, проверка, организованная с помощью ключевого слова assert, должна пройти без проблем. Но этому не бывать из-за отсутствия рефлексивности, что противоречит гипотезе.

Кроме перечисленных ограничений есть еще одно: поведение метода opCmp должно быть согласовано с поведением метода opEquals:

```
// Отношение к opEquals
if (x == y) assert(x <= y && y <= x);
```

Отношение к opEquals определено не слишком строго: вполне вероятна ситуация, когда для двух классов неравенства x <= y и y <= x истинны одновременно – здравый смысл продиктовал бы, что значения x и y равны. Тем не менее вовсе не обязательно, что x == y. Простым примером может послужить класс, определяющий равенство в терминах регистрочувствительных строк, а упорядочивание – в терминах строк, нечувствительных к регистру.

6.8.5. static Object factory (string className)

Это любопытный метод, позволяющий создавать объект по заданному имени его класса. Класс, участвующий в этой операции, должен иметь конструктор без аргументов; иначе метод factory порождает исключение. Посмотрим на factory в действии.

```
// Файл test.d
import std.stdio;

class MyClass {
   string s = "Здравствуй, мир!";
}
```

```
void main() {
   // Создать экземпляр класса Object
   auto obj1 = Object.factory("object.Object");
   assert(obj1);
   // Теперь создать экземпляр класса MyClass
   auto obj2 = cast(MyClass) Object.factory("test.MyClass");
   writeln(obj2.s); // Writes "Hello, world!"
   // factory с именем несуществующего класса возвращает null
   auto obj3 = Object.factory("Несуществующий");
   assert(!obj3);
}
```

Возможность создавать объект по строке очень полезна для реализации множества идей, таких как шаблон проектирования «Фабрика» [27, глава 23] и сериализация объекта. Казалось бы, в записи

```
void widgetize() {
   Widget w = new Widget;
    ../* Использование w */...
}
```

нет ничего плохого. Однако позже вы, возможно, передумаете и решите, что для текущей задачи лучше подходит TextWidget, класс-наследник, значит, предыдущий код придется привести к следующему виду:

```
void widgetize() {
   Widget w = new TextWidget;
     ./* Использование w */.
}
```

Проблема в том, что придется *изменить код*. Хирургическое вмешательство в код ради внесения нового функционала – зло, поскольку, поступая так, вы сильно рискуете испортить уже имеющуюся функциональность. В идеале для добавления функциональности нужно только добавлять код; это позволяет надеяться, что существующий код продолжит работать как обычно. Это как раз тот случай, когда особенно ценны функции, которые можно переопределять, ведь они позволяют настраивать код, не изменяя его, а лишь внося свои дополнения в особых выверенных точках. Бертран Мейер в шутку по-дзэнски назвал это *принципом Открытости/Закрытости* [40]: класс (единица инкапсуляции, в более общей формулировке) должен быть открыт для расширений, но закрыт для изменений. Оператор new работает с точностью до наоборот – требует, чтобы вы изменили инициализацию объекта w, если хотите корректировать его поведение. Гораздо лучшим решением послужила бы передача имени класса снаружи, ведь таким образом функция widgetize отделяется от определенного выбранного виджета:

```
void widgetize(string widgetClass) {
   Widget w = cast(Widget) Object.factory(widgetClass);
    /* Использование w */.
}
```

Теперь функция widgetize освобождена от ответственности за выбор конкретного класса из цепочки наследования, которую образует класс Widget. Есть и другие пути достижения гибкости конструирования объектов, расширяющие пространство проектирования в разных направлениях. Чтобы познакомиться с всеобъемлющим описанием этой проблемы, внимательно прочтите статью с драматическим названием «Java's new considered harmful» (Оператор new из Java опасен) [4].

6.9. Интерфейсы

Обычно объект (экземпляр класса) содержит состояние и определяет методы, работающие с этим состоянием. Выходит, объект одновременно действует по отношению к внешнему миру и как интерфейс (через свои общедоступные методы), и как инкапсулированная реализация этого интерфейса.

Тем не менее иногда полезно разграничить понятия интерфейса и реализации. Особенно полезно, когда требуется определить взаимодействие разнообразных частей большой программы. Функцию, оперирующую экземпляром класса Widget, должен интересовать исключительно интерфейс этого класса, а его реализация – по определению инкапсуляции – рассмотрению не подлежит. Таким образом, на первый план выносится понятие совершенно абстрактного набора методов, состоящего только из методов, которые класс *должен* реализовать, но лишенного всякой реализации. Такую сущность и называют *интерфейсом*.

Определение интерфейса в D выглядит почти как определение класса. Но кроме замены ключевого слова class на interface для него еще действуют определенные ограничения. В интерфейсе нельзя определять нестатические данные и реализовывать функции, допускающие переопределение. Внутри интерфейса разрешается определять статические данные и финальные функции с реализацией. Например:

```
interface Transmogrifier {
   void transmogrify();
   void untransmogrify();
   final void thereAndBack() {
      transmogrify();
      untransmogrify();
   }
}
```

Этого достаточно, чтобы функция, использующая Transmogrifier, скомпилировалась. Например:

```
void aDayInLife(Transmogrifier device, string mood) {
   if (mood == "играть") {
      device.transmogrify();
      play();
      device.untransmogrify();
```

```
    } else if (mood == "экспериментировать") {
        device.thereAndBack();
    }
}
```

Разумеется, поскольку пока еще нет определений для элементов интерфейса Transmogrifier, разумного способа вызвать функцию aDayInLife тоже нет. Поэтому давайте создадим реализацию этого интерфейса:

```
class CardboardBox : Transmogrifier {
    override void transmogrify() {
        // Залезть в коробку

    }
    override void untransmogrify() {
        // Вылезти из коробки

    }
}
```

Для реализации интерфейса используется тот же синтаксис, что и в случае реализации обычного наследника. Класс CardboardBox позволяет инициировать такой вызов:

```
aDayInLife(new CardboardBox, "играть");
```

Любая реализация интерфейса является подтипом этого интерфейса, так что она автоматически конвертируется в него. Мы воспользовались этим механизмом, просто передав объект CardboardBox вместо интерфейса Transmogrifier, ожидаемого функцией aDayInLife.

6.9.1. Идея невиртуальных интерфейсов (NVI)

Кое-что здесь может показаться странным: в интерфейсе Transmogrifier есть финальная функция. А как же возвышенная поэзия абстрактной, нереализованной функциональности? Ведь абстрактный интерфейс не должен определять реализацию!

В 2001 году Герб Саттер в своей статье [52] выдвинул интересный тезис, который позже развил в книге [55, пункт 39]. Определяемые интерфейсом методы, которые можно переопределять (такие как transmogrify и untransmogrify в нашем примере), играют две роли. Во-первых, они являются элементами самого интерфейса, то есть теми функциями, которые будет вызывать клиентский код, чтобы выполнить свою задачу. Во-вторых, такие методы также служат точками для внесения изменений, ведь именно их напрямую переопределяют классы-наследники. Как отметил Саттер, может оказаться полезным разделить методы интерфейса на две категории: абстрактные низкоуровневые методы, которые позже нужно будет реализовать, *плюс* высокоуровневые, видимые методы, которые может использовать клиентский код. Эти два множества могут пересекаться, а могут и не пересекаться, но считать их одинаковыми было бы значительной потерей.

У разделения методов на те, что видит клиентская сторона, и те, что определяет сторона реализующая, есть много достоинств. Такой подход позволяет разрабатывать интерфейсы, дружественные к реализации и к использованию одновременно. Интерфейсу, удовлетворяющему как нуждам реализации, так и нуждам клиентов, нужен компромисс между всеми предъявляемыми к нему требованиями. Слишком много внимания к реализации ведет к банальным, многословным, низкоуровневым интерфейсам, провоцирующим дублирование в клиентском коде, а слишком много внимания к клиентскому коду порождает большие, свободные, избыточные интерфейсы, определяющие помимо необходимых примитивов дополнительные функции, введенные для удобства пользователя. Идея невиртуальных[1] интерфейсов (NVI) позволяет облегчить жизнь обеим сторонам. Так, интерфейс `Transmogrifier` предоставляет пользователям дополнительную функцию, которая для удобства введена как метод `thereAndBack`, определенный в терминах примитивных операций.

Нарождающаяся идея напоминала шаблон проектирования «Шаблонный метод», но все же казалась достаточно уникальной, чтобы обрести собственное имя – невиртуальный интерфейс (Non-Virtual Interface, NVI). К сожалению, несмотря на то что NVI со временем превратился в популярный шаблон проектирования, по статусу он оставался скорее соглашением между хорошими разработчиками, не достигнув уровня средства языка, позволяющего гарантировать постоянство разработки. Недостаточная поддержка NVI языками в основном связана с тем, что он появился уже после того, как были определены популярные языки программирования, способствовавшие лучшему пониманию техники ООП, которое и привело к возникновению NVI. Так что язык Java вообще не поддерживает NVI, C# поддерживает весьма ограниченно (хотя широко использует NVI в качестве руководящей идеи для разработки), а C++ предоставляет хорошую поддержку на основе соглашений, но при этом не дает серьезных гарантий ни вызывающему, ни реализующему коду.

D полностью поддерживает NVI, предоставляя особые гарантии, если интерфейсы используют спецификаторы доступа. Рассмотрим пример. Предположим, автор интерфейса `Transmogrifier` сильно обеспокоен тем, что его интерфейс могут некорректно использовать: что если метод `transmogrify` вызовут, а метод `untransmogrify` забудут? Давайте запретим клиентам использовать все, кроме метода `thereAndBack`, а реализацию обяжем определить методы `transmogrify` и `untransmogrify`:

```
interface Transmogrifier {
   // Клиентский интерфейс
   final void thereAndBack() {
      transmogrify();
```

[1] Виртуальный метод – метод, который переопределяет другой метод или сам может быть переопределен. – *Прим. науч. ред.*

```
        untransmogrify();
    }
    // Интерфейс реализации
    private:
        void transmogrify();
        void untransmogrify();
}
```

Интерфейс Transmogrifier закрыл два своих внутренних элемента. Такие настройки определяют любопытную структуру и поведение программы: класс, реализующий интерфейс Transomgrifier, должен определять методы transmogrify и untransmogrify, но не имеет права их вызывать. На самом деле, никто не сможет вызвать эти две функции извне модуля Transmogrifier. Единственный способ вызвать их – неявный вызов через высокоуровневый метод thereAndBack, в чем, собственно, и состоит цель разработки: тщательно определенные точки доступа и хорошо структурированный поток управления между вызовами, адресующими эти точки. Язык пресекает обычные попытки нарушить эти гарантии. Например, реализующий класс не может ослабить уровень защиты методов transmogrify и untransmogrify:

```
class CardboardBox : Transmogrifier {
    override private void transmogrify() {     } // Все в порядке
    override void untransmogrify() { .. }       // Ошибка!
        // Нельзя изменить уровень защиты метода untransmogrify
        // с закрытого на общедоступный!
}
```

Разумеется, поскольку это все же ваша реализация, вы можете сделать метод общедоступным, если захотите, но придется дать ему другое имя:

```
class CardboardBox : Transmogrifier {
    override private void transmogrify() {     } // Все в порядке
    override private void untransmogrify() {     // Все в порядке
        doUntransmogrify();
    }
    void doUntransmogrify() {     }             // Все в порядке
}
```

Теперь пользователи класса CardboardBox могут вызывать метод doUntransmogrify, который делает ровно то же самое, что и метод untransmogrify. Но важно то, что метод void untransmogrify() именно с этими именем и сигнатурой реализующий класс показать не может. Таким образом, клиентскому коду никогда не будет доступна закрытая функциональность с закрытым именем. Если реализация захочет определить и документировать дублирующую функцию, это ее право.

Второй принцип, с помощью которого D гарантирует постоянство реализации NVI, – запрет переопределения финальных методов: никакая реализация интерфейса Transmogrifier не может определить метод, который бы мог успешно переопределить thereAndBack. Например:

```
class Broken : Transmogrifier {
   void thereAndBack() {
      // Почему бы не сделать это дважды?
      this.Transmogrifier.thereAndBack();
      this.Transmogrifier.thereAndBack();
   }
   // Ошибка! Нельзя переопределить
   // финальный метод Transmogrifier.thereAndBack

}
```

Если бы такое перекрытие было разрешено, то клиент, знающий, что класс Broken реализует интерфейс Transmogrifier, не смог бы без колебаний сделать вызов obj.thereAndBack() применительно к объекту obj типа Broken; не было бы никакой уверенности в том, что метод thereAndBack делает то, что предписано и документировано интерфейсом Transmogrifier. Конечно, клиентский код мог бы вызвать obj.Transmogrifier.thereAndBack(), таким образом гарантируя, что вызов пойдет в нужном направлении, но подобные решения, основанные на сверхвнимательности, мало кого привлекут. В конце концов хороший код не ждет, пока вы ослабите бдительность, а просто вдруг начинает вести себя странно. И последнее: если интерфейс определяет общедоступную функцию, она остается видимой во всех его реализациях. Если реализация также является финальной, у реализующего класса нет способа перехватить вызов. Реализация при этом может определить функцию с тем же именем, если это не вызовет конфликта. Например:

```
class Good : Transmogrifier {
   void thereAndBack(uint times) {
      // Почему бы не сделать это несколько раз?
      foreach (i; 0    times) {
         thereAndBack();
      }
   }

}
```

Этот код корректен, потому конфликт невозможен: вызов будет выглядеть либо как obj.thereAndBack() — и направится к методу Transmogrify.thereAndBack, либо как obj.thereAndBack(n) — и направится к методу Good.thereAndBack. В частности, реализация Good.thereAndBack не обязана относить свой внутренний вызов к реализации одноименной функции интерфейса.

6.9.2. Защищенные примитивы

Иногда, сделав функцию интерфейса закрытой, мы накладываем больше ограничений, чем требуется. Например, следствием такого шага является запрет вызова функции super со стороны реализации:

```
import std.exception;

class CardboardBox : Transmogrifier {
private:
   override void transmogrify() {     }
   override void untransmogrify() {     }
}
class FlippableCardboardBox : CardboardBox {
private:
   bool flipped;
   override void transmogrify() {
      enforce(!flipped, "Невозможно вызвать метод transmogrify: "
         "коробка работает в режиме машины времени");
      super.transmogrify(); // Ошибка! Нельзя вызвать
                            // закрытый метод CardboardBox.transmogrify!
   }
}
```

Когда коробка перевернута (flipped), она не может действовать как трансмогрификатор, поскольку – как всем известно – в этом случае она всего лишь скучная машина времени. И класс FlippableCardboardBox гарантирует такое поведение с помощью вызова функции enforce. Но если коробка не перевернута, то новый класс не сможет обратиться к версии метода transmogrify своего родителя. Что же делать?

Одно из возможных решений – фокус с переименованием, рассмотренный ранее на примере функции doUntransmogrify, но такая практика очень скоро надоедает, если приходится применять ее для нескольких методов. Более простое решение – ослабить уровень защиты двух открытых для переопределения методов класса Transmogrifier, заменив спецификатор доступа private на protected:

```
interface Transmogrifier {
   final void thereAndBack() {     }
protected:
   void transmogrify();
   void untransmogrify();
}
```

Защищенный уровень доступа позволяет реализации обращаться к родительской реализации. Заметим, что усиливать защиту так же некорректно, как и ослаблять. Если интерфейс определил метод, реализация не может наложить на него более строгие ограничения для обеспечения защиты. Например, при условии что интерфейс Transmogrifier определяет оба метода transmogrify и untransmogrify как защищенные, этот код окажется ошибочным:

```
class BrokenInTwoWays : Transmogrifier {
   public void transmogrify() {     }     // Ошибка!
   private void untransmogrify() {     }  // Ошибка!
}
```

Технически осуществимо как ослабление, так и ужесточение требований интерфейса к реализации, но эти возможности вряд ли можно использовать во благо. Интерфейс выражает цель, и должно быть достаточно ознакомиться лишь с определением интерфейса, чтобы полноценно использовать его независимо от доступности статического типа реализуемого класса.

6.9.3. Избирательная реализация

Иногда два интерфейса определяют общедоступные финальные методы с одинаковой сигнатурой, что приводит к двусмысленности:

```
interface Timer {
   final void run() {     }

}
interface Application {
   final void run() {     }

}
class TimedApp : Timer, Application {
       // Невозможно определить метод run()
}
```

В подобных случаях класс `TimedApp` не может определить собственный метод `run()`, поскольку таким образом он попытался бы незаконно перехватить сразу два метода, а на двух стульях, как известно, не усидишь. Но избавление от одного из двух ключевых слов `final` (в интерфейсе `Timer` или в интерфейсе `Application`) ситуацию бы не спасло, поскольку одно переопределение все равно оставалось бы в силе. Вот если бы оба метода были виртуальными, то мы бы выиграли: метод `TimerApp.run` реализовывал бы методы `Timer.run` и `Application.run` *одновременно*.

Чтобы получить доступ к этим методам объекта `app` типа `TimedApp`, вам придется написать `app.Timer.run()` и `app.Application.run()` для версий интерфейсов `Timer` и `Application` соответственно. Класс `TimedApp` может определить собственные функции, которые будут делегировать вызов этим методам. Главное, чтобы такие функции не пытались подменить `run()`.

6.10. Абстрактные классы

Нередко бывает, что родительский класс не в состоянии предоставить какую-либо разумную реализацию для некоторых или даже всех своих методов. Можно было бы преобразовать этот класс в интерфейс, но иногда удобно, что такой класс определяет некоторое состояние и виртуальные методы (привилегии, не доступные интерфейсам). И тут на помощь приходят абстрактные классы: они почти такие же, как и обычные, отличие лишь в том, что им дозволено оставлять функции нереализованными – достаточно лишь объявить их с ключевым словом `abstract`.

В качестве иллюстрации рассмотрим проверенный временем пример с иерархией объектов-фигур, задействованных в векторном графическом редакторе. В основании иерархии – класс Shape. Любая фигура обладает ограничивающим ее прямоугольником, так что, возможно, класс Shape захочет определить его в качестве своего поля (интерфейс не смог бы этого сделать). С другой стороны, некоторые методы класса Shape (такие как draw) должны остаться нереализованными, поскольку он не может реализовать их осмысленно. Считается, что эти методы определят потомки класса Shape.

```
class Rectangle {
   uint left, right, top, bottom;
}

class Shape {
   protected Rectangle _bounds;
   abstract void draw();
   bool overlaps(Shape that) {
      return _bounds.left <= that._bounds.right &&
      _bounds.right >= that._bounds.left &&
      _bounds.top <= that._bounds.bottom &&
      _bounds.bottom >= that._bounds.top;
   }
}
```

Метод draw – абстрактный, что означает три вещи. Во-первых, компилятор не ожидает от класса Shape реализации метода draw. Во-вторых, компилятор запрещает создавать экземпляры класса Shape. В-третьих, компилятор запрещает создавать экземпляры любых наследников класса Shape, не реализующих (явно или неявно, благодаря предку) метод draw. Слова «явно или неявно» означают, что требование реализации не является транзитивным; например, если определить наследника класса Shape с именем RectangularShape, который реализует метод draw, то заново реализовывать этот метод в потомках RectangularShape необязательно.

Компилятор *не ожидает* реализации абстрактного метода, но это не значит, что ее запрещается предоставлять. Например, вполне корректно предоставить реализацию метода Shape.draw. Клиентский код может вызывать этот метод, явно квалифицируя вызов, как здесь: this.Shape.draw().

Интересно то, что метод overlaps одновременно реализован и открыт для переопределения. По умолчанию он дает приблизительный ответ на вопрос о пересечении двух фигур, понимая под пересечением фигур пересечение их ограничивающих прямоугольников. Из-за такой трактовки этот метод работает неточно применительно к большинству непрямоугольных фигур; например, два круга могут и не перекрывать друг друга, даже если их ограничивающие прямоугольники пересекаются.

Класс, обладающий хотя бы одним абстрактным методом, и сам называется *абстрактным*. Если класс RectangularShape является наследником

абстрактного класса Shape и не переопределяет все абстрактные методы класса Shape, то класс RectangularShape также считается абстрактным и передает требование реализовать эти абстрактные методы по наследству своим потомкам. Вдобавок классу RectangularShape разрешается вводить новые абстрактные методы. Например:

```
class Shape {
   // Как и ранее
   abstract void draw();

}
class RectangularShape : Shape {
   // Наследует один абстрактный метод от класса Shape
   // и вводит еще один
   abstract void drawFrame();
}
class ARectangle : RectangularShape {
   override void draw() {     }
   // Класс ARectangle все еще абстрактен
}
class SolidRectangle : ARectangle {
   override void drawFrame() {     }
   // Класс SolidRectangle конкретен:
   // больше не осталось нереализованных абстрактных функций
}
```

Самое интересное, что класс может решить заново объявить функцию абстрактной, даже если до него ее уже переопределили и реализовали! В следующем фрагменте кода определен абстрактный класс, от него порожден конкретный класс, а затем от конкретного класса снова порождается абстрактный класс – и все из-за одного-единственного метода.

```
class Abstract {
   abstract void fun();
}
class Concrete : Abstract {
   override void fun() {     }
}
class BornAgainAbstract : Concrete {
   abstract override void fun();
}
```

Можно сделать конечной реализацию абстрактного метода...

```
class UltimateShape : Shape {
   // Вот и все о методе draw
   override final void draw() {     }
}
```

...но, по понятным причинам, нельзя определить метод одновременно и абстрактный, и финальный.

Если нужно объявить целую группу абстрактных методов, то можно использовать одну и ту же запись abstract несколько раз, как и спецификатор доступа (см. раздел 6.7.1):

```
class QuiteAbstract {
   abstract {
      // В этом контексте все абстрактное
      void fun();
      int gun();
      double hun(string);
   }
}
```

Абстрактность никак нельзя «выключить» внутри блока abstract, поэтому следующее определение некорректно:

```
class NiceTry {
   abstract {
      void fun();
      final int gun(); // Ошибка!
      // Определять финальные абстрактные функции нельзя!
   }
}
```

Ключевое слово abstract можно использовать и в виде метки:

```
class Abstractissimo {
   abstract:
      // Ниже все абстрактное
      void fun();
      int gun();
      double hun(string);
}
```

Применив однажды метку abstract:, ее действие невозможно «выключить».

Наконец, можно пометить с помощью abstract целый класс:

```
abstract class AbstractByName {
   void fun() {}
   int gun() {}
   double hun(string) {}
}
```

В свете постепенного усиления приведенных вариантов ключевого слова abstract может показаться, что, сделав абстрактным целый класс, можно нанести удар и посерьезнее, сделав абстрактным каждый отдельный метод. Ничего подобного. Грубость действия такого средства исключила бы возможность его употребления. Ключевое слово abstract перед классом просто запрещает клиентскому коду создавать экземпляры этого класса – можно создавать только экземпляры его неабстрактных потомков. Продолжим начатый выше пример с классом AbstractByName:

```
unittest {
   auto obj = new AbstractByName; // Ошибка! Нельзя создать
      // экземпляр абстрактного класса AbstractByName!
}
class MakeItConcrete : AbstractByName {
}
unittest {
   auto obj = new MakeItConcrete; // OK
}
```

6.11. Вложенные классы

Вложенные классы – интересное средство языка, заслуживающее особого внимания. Они полезны в качестве строительного материла для реализации более важных идей, таких как множественное порождение подтипов (которое рассматривается ниже).

Класс может определять другой класс прямо внутри себя:

```
class Outer {
   int x;
   void fun(int a) {     }
   // Определить внутренний класс
   class Inner {
      int y;
      void gun() {
         fun(x + y);
      }
   }
}
```

Вложенные классы – это просто обычные... *Минуточку*, как получилось, что метод Inner.gun обладает доступом к нестатическим полям и методам класса Outer? Если бы Outer.Inner было классическим определением класса в контексте класса Outer, из него было бы невозможно обращаться к данным и вызывать методы объекта Outer. В самом деле, откуда появляется доступ к этому объекту? Давайте просто создадим объект типа Outer.Inner и посмотрим, что произойдет:

```
unittest {
   // Сработать не должно
   auto obj = new Outer.Inner;
   obj.gun(); // Тут наступит конец света, поскольку
              // в поле зрения нет ни Outer.x, ни Outer.fun -
              // здесь вообще нет никакого Outer!
}
```

Поскольку в этом коде создается лишь объект типа Outer.Inner, а о создании экземпляра класса Outer речь не идет, единственные данные, под которые будет выделена память, – это данные, которые определяет класс Outer.Inner (то есть y), но не класс Outer (то есть x).

Удивительным образом определение класса действительно компилируется, а тест модуля – нет. Что же происходит?

Начнем с того, что вы никогда не сможете создать объект класса Inner, не имея в своем распоряжении экземпляр класса Outer. Это ограничение очень осмысленно, учитывая что Inner обладает магическим доступом к состоянию и методам Outer. Вот пример корректного создания объекта типа Outer.Inner:

```
unittest {
   Outer obj1 = new Outer;
   auto obj = obj1.new Inner; // Ага!
}
```

Сам синтаксис выражения new указывает на то, что происходит: для создания объекта типа Outer.Inner необходимо, чтобы уже существовал объект типа Outer. Ссылка на этот объект (в нашем случае obj1) неявно сохраняется в объекте типа Inner в качестве значения свойства outer, определенного на уровне языка. Затем, когда бы вы ни решили воспользоваться внутренним элементом класса Outer (таким как x), компилятор переписывает обращение к нему (в случае x получится запись this.outer.x). Инициализация сохраняемой во вложенном объекте скрытой ссылки на внешний контекст[1] происходит прямо перед вызовом конструктора этого вложенного объекта, так что у самого конструктора доступ ко внутренним элементам внешнего объекта уже есть. Наконец, протестируем все это, внеся несколько изменений в пример с классами Outer и Inner:

```
class Outer {
   int x;
   class Inner {
      int y;
      this() {
         x = 42;
         // x - то же, что this.outer.x
         assert(this.outer.x == 42);
      }
   }
}
unittest {
   auto outer = new Outer;
   auto inner = outer.new Inner;
   assert(outer.x == 42); // Вложенный объект inner
                          // изменил внешний объект outer
}
```

Если вы создаете объект типа Outer.Inner из нестатической функции-члена класса Outer, нет необходимости располагать перед оператором new префикс this – это делается неявно. Например:

[1] Это напоминает виртуальное наследование в C++. – *Прим. науч. ред.*

```
class Outer {
   class Inner {     }
   Inner _member;
   this() {
      _member = new Inner;          // То же, что this.new Inner
      assert(member.outer is this); // Проверить связь
   }
}
```

6.11.1. Вложенные классы в функциях

Как ни странно, вложение класса в функцию работает почти так же, как и вложение класса в другой класс. Класс, расположенный внутри функции, может обращаться к ее параметрам и локальным переменным:

```
void fun(int x) {
   string y = "Здравствуй";
   class Nested {
      double z;
      this() {
         // Обратиться к параметру
         x = 42;
         // Обратиться к локальной переменной
         y = "мир";
         // Обратиться к собственной переменной-члену
         z = 0.5;
      }
   }
   auto n = new Nested;
   assert(x == 42);
   assert(y == "мир");
   assert(n.z == 0.5);

}
```

Классы, вложенные в функции, особенно полезны, когда требуется корректировать поведение некоторого класса: имея функцию, возвращающую экземпляр этого класса, нужного наследника можно создать внутри нее. Приведем пример:

```
class Calculation {
   double result() {
      double n;

      return n;
   }
}

Calculation truncate(double limit) {
   assert(limit >= 0);
   class TruncatedCalculation : Calculation {
```

```
        override double result() {
            auto r = super.result();
            if (r < -limit) r = -limit;
            else if (r > limit) r = limit;
            return r;
        }
    }
    return new TruncatedCalculation;
}
```

Функция truncate переопределяет метод result класса Calculation: теперь этот метод возвращает значение в заданных пределах. В работе функции есть одна тонкость: обратите внимание на то, что переопределенный метод result использует параметр limit. Это не так уж странно, учитывая, что класс TruncatedCalculation используется внутри функции truncate, но ведь она возвращает экземпляр этого класса во внешний мир. Возникает простой вопрос: где остается значение параметра limit после того, как функция truncate вернет свой результат? В типичной ситуации компилятор помещает параметры и локальные переменные функции в стек, и после того как она вернула результат, они исчезают. Но в нашем примере значение параметра limit используется уже *после* того, как функция truncate вернула свой результат. То есть лучше бы параметру limit находиться где-то помимо стека, а иначе из-за небезопасного обращения к освобожденной памяти стека нарушится работа всего кода.

Но благодаря небольшой поддержке компилятора рассмотренный пример работает нормально. Когда бы компилятору ни приходилось компилировать функцию, он всегда просматривает ее в поисках нелокальных утечек (non-local escapes) – ситуаций, когда параметр или локальная переменная остается в использовании уже после того, как функция вернула результат[1]. Если обнаружена такая утечка, компилятор изменяет способ выделения памяти под локальное состояние (параметры плюс локальные переменные): вместо выделения памяти в стеке в этом случае происходит динамическое выделение памяти.

6.11.2. Статические вложенные классы

Посмотрим правде в глаза: вложенные классы – вовсе не то, чем кажутся. Мы воспринимаем их как обычные классы, определенные внутри классов или функций, но очевидно, что они необычны: особые синтаксис и семантика выражения new, магическое свойство .outer, другие правила поиска – вложенные классы определенно отличаются от обычных.

---

[1] Сходный механизм используется при возвращении функцией делегата и образовании замыканий, однако следует учитывать, что компилятор выполняет описанные действия строго для предопределенных языком случаев (таких как внутренние классы и делегаты). Попытка функции вернуть указатель на локальную переменную ни к чему хорошему не приведет. – *Прим. науч. ред.*

Что если требуется определить внутри класса или функции именно обычный класс? И вновь на помощь приходит ключевое слово static – достаточно поместить его перед определением этого класса. Например:

```
class Outer {
   static int s;
   int x;
   static class Ordinary {
      void fun() {
         writeln(s); // Все в порядке, доступ
                     // к статическому значению разрешен
         writeln(x); // Ошибка! Нельзя обратиться к нестатическому
                     // внутреннему элементу x!
      }
   }
}
unittest {
   auto obj = new Outer.Ordinary; // Все в порядке
}
```

Будучи обычным классом, статический внутренний класс не имеет доступа к внешнему объекту просто за отсутствием таковых. Зато благодаря контексту определения статический внутренний класс обладает доступом к статическим внутренним элементам внешнего класса.

6.11.3. Анонимные классы

Если в определении класса, заданном внутри спецификации суперкласса, отсутствуют имя и :, то это определение *анонимного класса*[1]. Такой класс всегда должен быть (нестатически) вложенным в функцию, и единственный способ использования этого класса – непосредственное создание его экземпляра:

```
class Widget {
   abstract uint width();
   abstract uint height();
}

Widget makeWidget(uint w, uint h) {
   return new class Widget {
      override uint width() { return w; }
      override uint height() { return h; }

   };
}
```

[1] Аргументы конструктора при создании анонимного класса передаются сразу после ключевого слова class, а если создается анонимный класс, реализующий список интерфейсов, то эти интерфейсы указываются через запятую после имени суперкласса. Пример: new class(arg1, arg2) BaseClass, Interface1, Interface2 {};. – *Прим. науч. ред.*

Это средство языка работает почти так же, как анонимные функции. Создание анонимного класса эквивалентно созданию нового именованного класса с последующим созданием его экземпляра. Эти два шага сливаются в один. Такое малопонятное средство может показаться бесполезным, но на практике многие проектные решения широко его используют для связи наблюдателей и субъектов [7].

6.12. Множественное наследование

D моделирует простое (одиночное) наследование классов и множественное наследование интерфейсов. Примерно так же поступают Java и C#, а такие языки, как C++ и Eiffel, идут другим путем.

Интерфейс может наследовать от любого числа интерфейсов. Поскольку он не может реализовать ни одну из функций, открытых для переопределения, интерфейс-наследник – это всего лишь усовершенствованный интерфейс, который требует реализации методов всех своих предков и, возможно, некоторых собственных. Пример:

```
interface DuplicativeTransmogrifier : Transmogrifier {
   Object duplicate(Object whatever);
}
```

Интерфейс DuplicativeTransmogrifier является наследником интерфейса Transmogrifier, так что теперь любой класс, реализующий интерфейс DuplicativeTransmogrifier, должен также реализовывать все методы интерфейса Transmogrifier, помимо только что объявленного метода duplicate. Отношение наследования работает как обычно: всюду, где ожидается интерфейс Transmogrifier, можно передавать интерфейс DuplicativeTransmogrifier, но не наоборот.

В общем случае интерфейс может стать наследником любого числа интерфейсов, как положено, накапливая примитивы, требующие реализации. Класс также может реализовать любое число интерфейсов. Например:

```
interface Observer {
   void notify(Object data);
   ...
}
interface VisualElement {
   void draw();

}
interface Actor {
   void nudge();

}
interface VisualActor : Actor, VisualElement {
   void animate();
```

```
}
class Sprite : VisualActor, Observer {
   void draw() {     }
   void animate() {     }
   void nudge() {     }
   void notify(Object data) {     }

}
```

На рис. 6.4 изображена только что закодированная иерархия наследования. Интерфейсы представлены овалами, а классы – прямоугольниками.

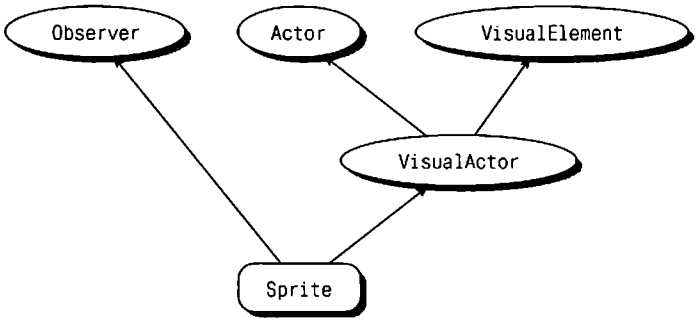

Рис. 6.4. Простая иерархия наследования, иллюстрирующая множественное наследование интерфейсов

Теперь определим класс Sprite2. Автор Sprite2 забыл, что интерфейс VisualActor уже является наследником интерфейса Actor, поэтому помимо интерфейсов Observer и VisualActor сделал родителем Sprite2 еще и интерфейс Actor. Полученная иерархия представлена на рис. 6.5.

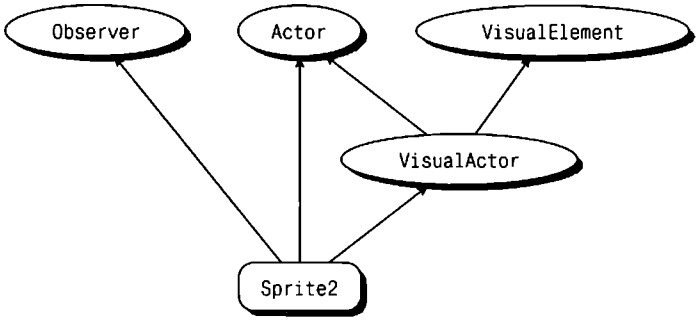

Рис. 6.5. Иерархия наследования с лишней ветвью (в данном случае это связь между Sprite2 и Actor). Удалять лишние связи необязательно, но большого труда это обычно не требует, а в результате получается более чистая структура и уменьшается размер объекта

Лишняя ветвь в иерархии тут же распознается как непосредственная связь с интерфейсом, от которого объект также наследует косвенно. Дублирующие связи не вызывают особых проблем, но в большинстве реализаций они увеличивают размер итогового объекта, в данном случае Sprite2.

Бывает, что один и тот же интерфейс наследуется через разные ветви, и ни одну из них удалить нельзя. Предположим, что сначала мы добавили интерфейс ObservantActor, наследующий от интерфейсов Observer и Actor:

```
interface ObservantActor : Observer, Actor {
   void setActive(bool active);
}
interface HyperObservantActor : ObservantActor {
   void setHyperActive(bool hyperActive);
}
```

Затем мы определили класс Sprite3, реализующий интерфейсы HyperObservantActor и VisualActor:

```
class Sprite3 : HyperObservantActor, VisualActor {
   override void notify(Object) {  .. }
   override void setActive(bool) {  .. }
   override void setHyperActive(bool) {     }
   override void nudge() {     }
   override void animate() {     }
   override void draw() {     }

}
```

Такие условия несколько меняют положение дел (рис. 6.6). Если Sprite3 хочет реализовать как HyperObservantActor, так и VisualActor, ему неизбежно придется реализовывать интерфейс Actor дважды (по разным связям). К счастью, для компилятора это не проблема – повторное наследование одного и того же интерфейса разрешено. Тем не менее повторное наследование одного и того же класса запрещено, поэтому и любое множественное наследование классов в D тоже запрещено.

Почему такая дискриминация? Что именно делает интерфейсы более податливыми для множественного наследования, чем классы? Подробное объяснение вышло бы довольно замысловатым, а если вкратце, то значимая разница между интерфейсом и классом в том, что последний может содержать состояние. И что гораздо важнее, класс может содержать модифицируемое состояние. Интерфейс же, напротив, не содержит собственного состояния; с каждым реализованным интерфейсом ассоциирована своя бухгалтерия (во многих реализациях это указатель на «виртуальную таблицу» – массив указателей на функции), но этот указатель идентичен для всех вхождений интерфейса в класс, никогда не меняется и находится под контролем компилятора. Компилятор использует эти ограничения с выгодой для себя: он размещает множество

копий этой «бухгалтерской» информации в классе, но класс об этом никогда не узнает.

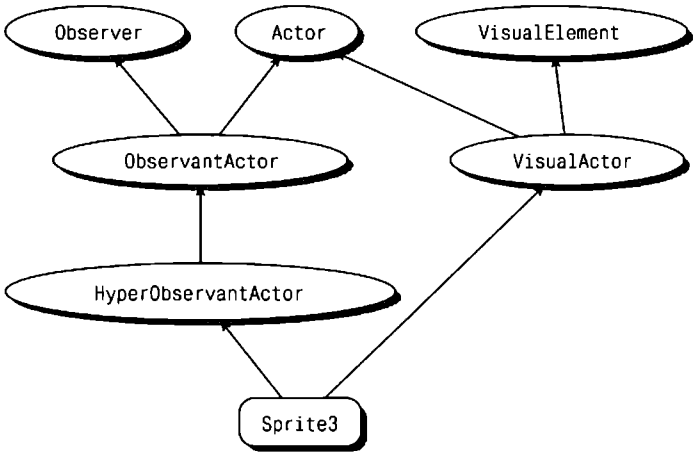

Рис. 6.6. Иерархия с множественными путями между узлами (в данном случае между узлами `Sprite3` и `Actor`). Такую структуру обычно называют «ромбовидной иерархией наследования», поскольку при отсутствии узла `HyperObservantActor` пути между `Sprite3` и `Actor` образовали бы ромб. В общем случае такие иерархии могут принимать разные формы. Их характерная особенность – множество путей от одного узла к другому

О достоинствах и недостатках множественного наследования спорят давно. Дебаты продолжаются и вряд ли прекратятся в ближайшем будущем, но в одном оппоненты сошлись: реализовать множественное наследование так, чтобы оно одновременно было простым, эффективным и полезным, непросто. Одни языки, придавая меньшее значение эффективности, делают выбор в пользу дополнительной выразительности, которую привносит множественное наследование. Другие языки, заложив основу высокой производительности (например, с помощью непрерывных объектов и скоростной диспетчеризации функций), ради этого ограничивают гибкость возможных программных решений. Интересное решение, позволяющее задействовать большинство преимуществ множественного наследования без свойственных ему проблем, – примеси в языке Scala (по сути, это интерфейсы, укомплектованные реализациями по умолчанию). Подход языка D заключается в разрешении множественного *порождения подтипов*, то есть порождения подтипов без наследования. Посмотрим, как это работает.

6.13. Множественное порождение подтипов

Продолжим достраивать программу, использующую класс Shape. Допустим, требуется определить объекты класса Shape, которые можно было бы хранить в базе данных. Мы нашли отличную библиотеку, которая обеспечивает постоянство объектов с помощью базы данных, что прекрасно нам подходит, кроме одного: она требует, чтобы каждый сохраняемый объект наследовал от класса DBObject.

```
class DBObject {
private:
   ... // Состояние
public:
   void saveState() {     }
   void loadState() {     }
   ..
}
```

Подобную ситуацию можно смоделировать по-разному, но согласитесь, если бы не ограничения языка, вполне естественно было бы определить класс StorableShape, который «являлся» бы классом Shape и DBObject одновременно. Основанием иерархии фигур в таком случае стал бы класс StorableShape. И любой объект типа StorableShape на экране выглядел и вел бы себя как объект типа Shape, а при переносе обратно в базу данных – как объект типа DBObject. Но это было бы множественное наследование классов, что в D *запрещено*, так что придется нам поискать альтернативное решение.

К счастью, язык приходит на помощь, предоставляя общий и очень полезный механизм – множественное порождение подтипов. Класс может указать, что он является подтипом любого другого класса, и для этого классу-подтипу не требуется становиться наследником класса-супертипа. Все, что нужно сделать, – указать объявление alias this. Простейший пример:

```
class StorableShape : Shape {
   private DBObject _store;
   alias _store this;
   this() {
      _store = new DBObject;
   }
   ..
}
```

Класс StorableShape наследует от и «является» классом Shape, но также является и классом DBObject. Когда бы ни потребовалось привести экземпляр класса StorableShape к экземпляру класса DBObject и когда бы ни выполнялся поиск элемента в классе StorableShape, поле _store тоже имеет право голоса. Запросы, сопоставляемые с DBObject, автоматически перенаправляются от this к this._store. Например:

```
unittest {
   auto s = new StorableShape;
   s.draw();        // Вызывает метод класса Shape
   s.saveState();   // Вызывает метод класса DBObject
                    // Перезаписывается в виде s._store.saveState()
   Shape sh = s;    // Обычное приведение "вверх" вида потомок -> предок
   DBObject db = s; // Перезаписывается в виде DBObject db = s._store
}
```

По сути, StorableShape – это подтип типа DBObject, а поле _store – это подобъект типа DBObject объекта типа StorableShape.

В классе может быть неограниченное число объявлений alias this, таким образом, класс может стать подтипом неограниченного числа типов[1].

6.13.1. Переопределение методов в сценариях множественного порождения подтипов

Но ведь не может все быть так просто, правда? И все непросто, потому что класс StorableShape все это время мошенничал. Да, обладая объявлением alias this, тип StorableShape номинально является типом DBObject, но не может напрямую переопределить ни один из методов класса DBObject. Очевидно, что методы класса Shape могут переопределяться как обычно, но где то место, где может быть переопределен метод DBObject.saveState? Возвращая _store в качестве псевдоподобъекта, мы уклоняемся от решения проблемы – на самом деле, ко внешнему объекту StorableShape привязано совсем немного информации о _store, по крайней мере, пока мы ничего не сделали, чтобы это изменить. Так посмотрим, что в наших силах.

Точное место, где смошенничало исходное определение класса StorableShape, – инициализация поля _store выражением new DBObject. Это полностью отделяет подобъект _store от внешнего объекта класса StorableShape, которому нужно переопределить методы DBObject. Итак, что нам необходимо сделать, так это определить новый класс MyDBObject внутри класса StorableShape. Этот класс сохранит обратную ссылку на внешний объект StorableShape и переопределит все методы, которые требуется переопределить. Наконец, внутри переопределенных методов класс MyDBObject обладает доступом ко всему классу StorableShape, и все можно делать так, будто в нашем распоряжении полноценное множественное наследование. Круто!

Если слова «внешний объект» напоминают вам что-то, уже прозвучавшее в этой главе, поздравляю: вы заметили одно из самых счастливых совпадений в анналах программирования. Вложенные классы (см. раздел 6.11) так хорошо подходят для «эмуляции» множественного насле-

[1] К сожалению, текущая на момент выхода книги версия компилятора допускала только одно объявление alias this. – *Прим. науч. ред.*

дования, что их можно принять за *deus ex machina*[1]. На самом деле, вложенные классы (на создание которых вдохновил язык Java) появились задолго до конструкции alias this.

Использование вложенных классов делает переопределение с alias this удивительно простым. Все, что нужно сделать в этом случае, – определить вложенный класс и сделать его наследником класса DBObject. Внутри такого класса можно переопределить какой угодно метод DBObject, и здесь вы обладаете полным доступом к общедоступным и защищенным определениям класса DBObject *и* ко всем определениям класса StorableShape. Если бы это было чуть легче, то было бы запрещено по крайней мере в нескольких штатах.

```
class StorableShape : Shape {
   private class MyDBObject : DBObject {
      override void saveState() {
      // Доступ к DBObject и StorableShape
      ..
      }
   }
   private MyDBObject _store;
   alias _store this;
   this() {
      // Вот решающий момент установления связи
      _store = this.new MyDBObject;
   }
   ...
}
```

Ключевым моментом является получение полем _store доступа ко внешнему объекту StorableShape. Как показано в разделе 6.11, создание вложенного класса позволяет чудесным образом сохранить внешний объект (в данном случае this) внутри вложенного класса. С помощью нотации this.new MyObject всего лишь предпринята попытка прояснить, что this обуславливает создание нового объекта MyDBObject. (На самом деле, this. и так подставляется неявно, поэтому в данном случае это можно не указывать.)

Единственным камнем преткновения служит то, что внутренние элементы DBObject будут перекрывать внутренние элементы StorableShape. Предположим, что оба класса DBObject и Shape определили поле с именем _name:

```
class Shape {
   protected string _name;
   abstract void draw();
```

[1] Deus ex machina («бог из машины») – в древнегреческом театре: бог, спускающийся с небес (изображающий его актер мог «летать» при помощи механического крана) и решающий все проблемы героев. В переносном смысле – неожиданная удачная развязка неразрешимой ситуации. – *Прим. ред.*

```
}
class DBObject {
   protected string _name;
   void saveState() {     }
   void loadState() {     }

}
```

При реализации множественного порождения подтипов с помощью класса `MyDBObject`**, вложенного в класс** `StorableShape`**, поле** `DBObject._name` **перекрывает поле** `StorableShape._name`**. Таким образом, если код внутри** `MyDBObject` **будет использовать просто** `_name`**, то через этот идентификатор он будет обращаться к** `DBObject._name`**.**

```
class StorableShape : Shape {
   private class MyDBObject : DBObject {
      override void saveState() {
         // Изменить Shape._name внешнего обьекта Shape
         this.outer._name = "A";
         // Изменить DBObject._name объекта-родителя
         _name = "B";
         // Просто для наглядности
         assert(super._name == "B");
      }
   }
   private MyDBObject _store;
   alias _store this;
   this() {
      _store = new MyDBObject;
   }

}
```

6.14. Параметризированные классы и интерфейсы

Иногда требуется параметризировать некоторый класс или интерфейс с помощью сущности, известной во время компиляции. Рассмотрим, например, определение интерфейса стека. Во избежание дублирования кода (`StackInt`**,** `StackDouble`**,** `StackWidget`**, ...) необходимо параметризировать этот интерфейс типом сохраняемых в стеке элементов. Определение параметризированного интерфейса в D:**

```
interface Stack(T) {
   @property bool empty();
   @property ref T top();
   void push(T value);
   void pop();
}
```

С помощью синтаксиса (T) в интерфейс Stack вводится параметр типа. В теле интерфейса T можно использовать так же, как любой другой тип. Обращаясь к интерфейсу Stack в клиентском коде, нужно указать аргумент. Такой аргумент можно передать с помощью бинарного оператора !, как здесь:

```
unittest {
   alias Stack!(int) StackOfInt;
   alias Stack!int SameAsAbove;
   ...
}
```

Там, где используется всего один параметр (как и в нашем случае с интерфейсом Stack), круглые скобки можно опустить.

Логично было бы реализовать интерфейс в классе. В идеале реализация тоже должна быть обобщенной (не должна быть настроена на какой-либо конкретный тип элементов). Следовательно, определяем параметризированный класс StackImpl, который принимает параметр T, передает его в Stack и использует внутри своей реализации. А теперь реализуем стек на основе массива:

```
import std.array;

class StackImpl(T) : Stack!T {
   private T[] _store;
   @property bool empty() {
      return _store.empty;
   }
   @property ref T top() {
      assert(!empty);
      return _store.back;
   }
   void push(T value) {
      _store ~= value;
   }
   void pop() {
      assert(!empty);
      _store.popBack();
   }
}
```

Работать с StackImpl так же весело, как и реализовывать его:

```
unittest {
   auto stack = new StackImpl!int;
   assert(stack.empty);
   stack.push(3);
   assert(stack.top == 3);
   stack.push(5);
   assert(stack.top == 5);
   stack.pop();
   assert(stack.top == 3);
```

```
    stack.pop();
    assert(stack.empty);
}
```

Как только вы создадите экземпляр параметризированного класса, он превратится в обычный класс, так что StackImpl!int – это такой же класс, как и любой другой. Именно этот конкретный класс реализует Stack!int, поскольку в формочку для вырезания StackImpl(T) под видом T вставили int по всему телу этого класса.

6.14.1. И снова гетерогенная трансляция

Теперь, раз уж мы заговорили о выведении настоящих типов из параметризированных типов, приглядимся к процессу подстановки. Впервые мы обсуждали понятие гетерогенной трансляции (которая противоположна гомогенной трансляции) в разделе 5.3 в контексте функций с обобщенными типами. Освежим в памяти основные моменты: в случае гомогенной трансляции инфраструктура языка предполагает, что все значения имеют один и тот же тип (например, все они объекты – Object), и автоматически подгоняет обобщенный (содержащий аргументы типов) код к этому общему типу. Подгонка может включать приведение типов в обе стороны, а также «упаковку» значений некоторых типов (чтобы заставить их соблюдать некоторый общий формат данных) и последующую их «распаковку» (когда пользовательский код захочет использовать упакованные значения). Этот процесс безопасен для типов и полностью прозрачен. Языки Java и C# используют гомогенную трансляцию для своих параметризированных типов.

В соответствии с гомогенным подходом все стеки StackImpl разделяют один и тот же код для реализаций своих методов. И, что более важно, на уровне типов не существует никаких различий: динамические типы StackImpl!int и StackImpl!double одинаковы. Транслятор, по сути, определяет *один* интерфейс для всех Stack!T и *один* класс для всех StackImpl!T. Такие типы называют *очищенными*, поскольку любая информация, специфичная для T, стирается. Затем транслятор искусно заменяет код, использующий Stack и StackImpl с разными T, чтобы использовались только упомянутые очищенные типы. Статическая информация о типах, которые клиентский код вставляет в Stack и StackImpl, надолго не сохраняется; эта информация служит для статической проверки типов, а затем сразу же забывается – или, лучше сказать, стирается. При таком подходе возникает ряд проблем – по той простой причине, что часть информации теряется. Вот простой пример: нельзя перегрузить функцию Stack!int функцией Stack!double и наоборот, поскольку у них один и тот же тип. Есть и более глубокие вопросы безопасности, подлежащие обсуждению. Отчасти они рассмотрены в научной литературе [14, 1, 49].

Гетерогенный транслятор (такой как механизм шаблонов C++) подходит к проблеме по-другому. Для гетерогенного транслятора Stack – не тип, а лишь средство для создания типа. (Еще одно иносказание для

ясности: тип – это схема значения, а параметризированный тип – схема типа.) Каждое воплощение Stack!int, Stack!string, стек любых других значений, которые могут понадобиться в вашем приложении, породит отдельный, отличный от других тип. Гетерогенный транслятор генерирует все эти типы, копируя и вставляя тело интерфейса Stack при замене T любым типом, который вы решили использовать с интерфейсом Stack. При таком подходе, скорее всего, будет сгенерировано больше кода, но все же это более мощное решение, поскольку статическая информация о типах сохраняется во всей полноте. Кроме того, при гетерогенной трансляции в каждом случае код генерируется индивидуально, следовательно, этот код может оказаться более быстрым.

D везде использует гетерогенную трансляцию, а это значит, что Stack!int и Stack!double – разные интерфейсы, а StackImpl!int и StackImpl!double – разные типы. Кроме общего происхождения от одного и того же параметризированного типа эти типы никак друг с другом не связаны. (Но вы, конечно, можете связать их некоторым образом, например, сделав все реализации интерфейса Stack наследниками общего интерфейса.) Поскольку класс StackImpl генерирует отдельную комбинацию методов для каждого подставленного в него типа, происходит частичное дублирование бинарного кода, что особенно раздражает, так как сгенерированный код часто оказывается полностью идентичным. Умный компилятор объединил бы все эти идентичные функции в одну (на момент написания этой книги D не обладает такими способностями, но в более зрелых компиляторах C++ такое объединение уже является принятой технологией).

Конечно же, у класса может быть больше одного параметра типа. Покажем это на примере класса StackImpl, наделив его одной интересной особенностью. Вместо того чтобы привязывать структуру хранения к массиву, вынесем это решение за пределы StackImpl. Из всей функциональности массивов StackImpl использует только empty, back, ~= и popBack. Так сделаем решение о выборе контейнера деталью реализации StackImpl:

```
class StackImpl(T, Backend) : Stack!T {
   private Backend _store;
   @property bool empty() {
      return _store.empty;
   }
   @property ref T top() {
      assert(!empty);
      return _store.back;
   }
   void push(T value) {
      _store ~= value;
   }
   void pop() {
      assert(!empty);
      _store.popBack();
   }
}
```

6.15. Переопределение аллокаторов и деаллокаторов[1]

Как мы помним, D является языком для системного программирования. Иногда бывает нужно воспользоваться преимуществами объектно-ориентированного программирования, применяя при этом собственную стратегию выделения памяти. Текущие реализации D любезно предоставляют возможность определить для класса собственные *аллокатор* (от англ. allocate – назначать, размещать) – функцию для выделения памяти и *деаллокатор* – функцию для ее освобождения. Вот как это выглядит:

```
import std.c.stdlib;
import core.exception;
class StdcClass {

   new(size_t obj_size) {
      void* ptr = malloc(obj_size);
      if (ptr is null)
         throw new OutOfMemoryError(__FILE__, __LINE__);
      return ptr;
   }

   delete(void* ptr) {
      if (ptr)
         free(ptr);
   }
}
```

Аллокатор объявляется как метод класса `new` без указания возвращаемого типа. Аллокатор всегда должен возвращать пустой указатель (`void*`). Аллокатор имеет минимум один аргумент – размер экземпляра класса в байтах типа `size_t`. В нашем примере память выделяется функцией `malloc` стандартной библиотеки языка C. Эта функция принимает в качестве аргумента размер выделяемой памяти и в случае успеха возвращает адрес выделенного блока памяти, иначе возвращает `null`. После вызова этой функции мы проверяем полученный адрес. Если он равен `null`, то порождаем исключение, в которое передаем имя текущего файла и номер текущей строки – информацию, полезную при отладке. Если указатель корректный, мы передаем его инструкции `return` аллокатора.

Деаллокатор объявляется с помощью ключевого слова `delete` и имеет один аргумент – указатель на блок памяти, в котором размещен объект.

---

[1] Описание этой части языка намеренно не было приведено в оригинале книги, но поскольку эта возможность присутствует в текущих реализациях языка, мы добавили ее описание в перевод. – *Прим. науч. ред.*

В приведенном примере, если указатель не пустой, мы освобождаем память, выделенную под объект.

Все. Мы получили класс, который не использует сборщик мусора. Теперь мы можем создавать и уничтожать экземпляры этого класса привычным для программистов на C++ образом:

```
StdcClass obj = new StdcClass();
...
delete obj;
```

Но будьте внимательны. Если вы уже успели расслабиться, полагаясь на сборщик мусора D, то соберитесь. Вам придется самостоятельно отслеживать созданные объекты и уничтожать их, когда в них пропадает необходимость, с помощью оператора delete. Каждому вызову new должен соответствовать вызов delete. Теперь от утечек памяти вас не спасет никто, кроме вас самих. Это плата за более быстрое и экономное создание объектов.

Но предположим, что мы предоставим пользователю класса право самому выделить память удобным для него образом и очистить ее также самостоятельно:

```
import std.c.stdlib;
import core.exception;
class StdcClass {

   new(size_t obj_size) {
      void* ptr = malloc(obj_size);
      if (ptr is null)
         throw new OutOfMemoryError(__FILE__, __LINE__);
      return ptr;
   }

   new(size_t, void* ptr) {
      if (ptr is null)
         throw new OutOfMemoryError(__FILE__, __LINE__);
      return ptr;
   }

   delete(void* ptr) {
      if (ptr)
         free(ptr);
   }
}
```

Попробуем его использовать.

```
static void[__traits(classInstanceSize, StdcClass)] data = void;

auto i_1 = new(data.ptr) StdcClass(); // Разместить объект
                                      // в статическом буфере
```

```
auto i_2 = new StdcClass(); // Выделить память встроенным аллокатором

delete i_2;
```

Как видим, аллокаторы могут быть перегружены обычным образом. Первый аргумент (размер экземпляра) обязателен, он передается конструктору автоматически. Остальные аргументы передаются выражению new. Деаллокатор перегружать нельзя.

Аллокаторы и деаллокаторы классов считаются устаревшей возможностью, и ее планируют убрать из самого языка, так как, не считая особого синтаксиса, ее можно запросто перевоплотить как простые функции в пользовательском коде. Например в стандартной библиотеке есть функция emplace:

```
import std.conv;
class C {
   int i;
   this() { i = 42; }
}
unittest {
   void[__traits(classInstanceSize, C)] data = void;
   auto c = emplace!C(data[]);
   assert(c.i == 42);
}
```

К тому же, как всегда, при большой силе – большая ответственность: если вы будете выделять память под классы, не используя кучу под контролем сборщика мусора, и ваш класс содержит указатели (в том числе неявные, унаследованные от классов-родителей) на данные, выделенные в памяти сборщика мусора, вы обязаны декларировать их создание сборщику мусора вызовом функции core.memory.GC.addRange(). Иначе сборщик мусора не будет знать о ссылках на данные, хранящихся в ваших классах, и может счесть мусором данные, на которые ваши классы все еще ссылаются. Это может повлечь за собой порчу данных и трудноуловимые ошибки программы, так как в случаях порчи данных сбой может произойти уже спустя пару минут после порчи в совершенно постороннем, невинном коде, который всего лишь пытался использовать свою ненароком испорченную область памяти. Мы вас предупредили!

6.16. Объекты scope[1]

В C++ есть возможность создать экземпляр класса как автоматическую переменную. При создании экземпляра память выделяется в стеке и запускается конструктор, а после выхода экземпляра из области видимости автоматически запускается деструктор.

[1] Описание этой части языка намеренно не было приведено в оригинале книги, но поскольку эта возможность присутствует в текущих реализациях языка, мы добавили ее описание в перевод. – *Прим. науч. ред.*

Эта возможность полезна при использовании небольших классов, создаваемых для выполнения какой-то одной задачи, выполнив которую, они становятся не нужны. При использовании автоматических переменных нет необходимости вызывать деструктор, поскольку он запускается автоматически. Такая техника в языках ООП называется RAII (Resource Acquisition Is Initialization – «получение ресурса есть его инициализация»), так как она относится и к инициализации при создании объекта, и к его уничтожению при покидании области видимости.

Для поддержки RAII язык D предоставляет структуры с деструкторами, которые будут описаны в следующей главе, и похожую конструкцию scope(exit). Но еще до того как D стал поддерживать структуры с деструкторами, подход RAII был реализован в объектах scope. Эта возможность присутствует и в текущих реализациях, но считается устаревшей и небезопасной.

По сути, объект scope – это класс памяти, и используется примерно так же:

```
scope obj = new SomeClass(arg_1);
scope SomeClass obj_2 = new SomeClass(arg_1);
```

Обе записи делают одно и то же. Они создают временный экземпляр класса, который автоматически будет уничтожен при выходе из области видимости. Память под него может быть выделена в стеке (а может, и нет), но в любом случае нельзя передавать ссылку за пределы области видимости переменной. При объявлении собственных аллокатора и деаллокатора будут использованы они. Гарантируется, что сразу после выхода из области видимости будет вызвана инструкция delete.

В отличие от C++, в D объект scope сохраняет ссылочную семантику.

Как следствие того, что память выделяется в стековом фрейме функции, объект scope не может быть возвращен функцией. Если это делается явно, компилятор генерирует ошибку:

```
Object oo() {
   scope t = new Object;
   return t;  // Ошибка времени компиляции
}
```

Однако если объект scope возвращается неявно, отследить это компилятор не сможет:

```
Object foo()
{
   scope t = new Object;
   Object p = t;
   return p; // Компилятор пропустит это,
             // но результат ни к чему хорошему не приведет
}
```

Чтобы избежать подобных конфликтов, D вводит понятие *класса scope*. Экземпляры класса scope могут быть только объектами scope. Объявле-

ния не scope-ссылки на объект компилятор не допустит. Для объявления класса scope нужно добавить в его объявление атрибут scope. Атрибут scope является транзитивным, то есть все подклассы класса scope также являются классами scope.

Объявим абстрактный класс для нахождения дайджеста (хеш-суммы). Конкретная реализация, произведенная от данного класса, может реализовывать алгоритм нахождения MD5-дайджеста, SHA-256 или любого другого, но интерфейс от этого не изменится.

```
import std.stdio;
abstract scope class Digest
{
   abstract void update (void[] data);
   abstract ubyte[] result();
}

class Md5: Digest { // Тоже scope


}
```

Приведем пример функции для вычисления MD5-суммы какого-то файла:

```
ubyte[] calculateHash(string filename)
{
   scope digest = new Md5();
   auto fd = File(filename, "r");
   ubyte[] buff = new ubyte[4096];
   ubyte[] readBuff;
   do {
      readBuff = fd.rawRead(buff);
      digest.update(readBuff);
   }
   while (readBuff.length);
   return digest.result();
}

void main(string[] args)
{
   ubyte[] hash = calculateHash(args[1]);

}
```

Однако даже при использовании классов scope компилятор не может отследить сохранение ссылки на объект при передаче ее функции с неявным приведением типов:

```
scope class C { }

Object saved;
void save(Object c) {
```

```
    saved = c;
}

void derp() {
    scope c = new C;
    save(c);
}

void main() {
    derp();
    // Куда сейчас указывает глобальная переменная saved?
}
```

Использование объектов scope может повысить производительность. Если класс не переопределяет аллокатор по умолчанию, память под объект может быть выделена в стеке. Выделить память в стеке гораздо быстрее, чем динамическую память. Кроме того, объект scope использует память ровно столько, сколько нужно, автоматически освобождая ее. В случае же со сборщиком мусора память будет освобождена когда-нибудь. Чем больше памяти будет выделяться и освобождаться не через сборщик мусора, тем чаще будет производиться сбор. Во время сбора мусора приостанавливаются все потоки выполнения и производится поиск блоков памяти, на которые нет ни одной ссылки. Такие блоки памяти освобождаются.

Класс памяти scope и классы scope считаются устаревшими и небезопасными, и их планируют убрать из самого языка. Вместо них в стандартную библиотеку была добавлена очень похожая конструкция scoped!T:

```
import std.typecons;
unittest {
    class A { int x; }
    auto a1 = scoped!A();
    auto a2 = scoped!A();
    a1.x = 42;
    a2.x = 53;
    assert(a1.x == 42);
}
```

Конструкция scoped!T, которую можно найти в модуле std.typecons, столь же небезопасна, но не засоряет очередной излишней возможностью язык (осложняя этим жизнь авторам компиляторов D).

6.17. Итоги

Классы – это основное средство для реализации объектно-ориентированных решений в D. Они повсюду используют ссылочную семантику и утилизируются сборщиком мусора.

Наследование позволяет использовать динамический полиморфизм. Разрешено только одиночное наследование, но класс может стать наслед-

ником нескольких интерфейсов. Интерфейсы не обладают собственным состоянием, но могут определять финальные методы.

Правила защиты соответствуют правилам защиты, принятым в операционной системе (каталоги и файлы).

Все классы обладают общим предком – классом `Object`, определенным в модуле `object`, который является частью реализации D. Класс `Object` определяет несколько важных примитивов, а модуль `object` – функции, на которые опирается сравнение объектов.

Класс может определять вложенные классы, автоматически сохраняющие ссылку на свой внешний класс, и статические вложенные классы, которые не сохраняют ссылку на внешний класс.

D полностью поддерживает технику невиртуальных интерфейсов, а также полуавтоматический механизм для множественного порождения подтипов.

7

Другие пользовательские типы

Применяя классы, основные типы и функции, можно написать много хороших программ. С параметризированными классами и функциями дело идет еще лучше. Но нередко мы с сожалением отмечаем, что по нескольким причинам классы не представляют собой инструмент с максимальной абстракцией типа.

Во-первых, классы подчиняются ссылочной семантике и из-за этого могут воплощать многие проектные решения не полностью или с ощутимыми накладными расходами. На практике трудно моделировать с помощью класса такую простую сущность, как точка с двумя или тремя координатами, если таких точек больше нескольких миллионов: разработчик оказывается перед непростым выбором – хорошая абстракция или приемлемое быстродействие. Кроме того, для линейной алгебры ссылочная семантика – большая морока. Попробуйте убедить математика или программиста-теоретика, что присваивание `a = b` должно делать из матрицы `a` лишь псевдоним матрицы `b`, а не отдельную копию! Даже такой простой тип, как массив, довольно накладно моделировать в виде класса в сравнении с мощной и лаконичной абстракцией массива, имеющейся в языке D (см. главу 4). Можно, конечно, сделать массивы «волшебными», но опыт то и дело показывает, что предоставлять множество «волшебных» типов, не воспроизводимых в пользовательском коде, – дурной тон и признак плохо спроектированного языка. Затраты на массив – всего два слова, а выделение памяти под экземпляр класса и использование дополнительного косвенного обращения означают большие накладные расходы по памяти и времени для всех примитивов массива. Даже такой простой тип, как `int`, нельзя выразить в виде класса дешево и элегантно (причем речь не об удобстве оператора). У такого класса, как `BigInt`, та же проблема: `a = b` делает нечто совершенно иное, чем соответствующая операция присваивания для типа `int`.

Во-вторых, классы живут вечно, а значит, с их помощью трудно моделировать ресурсы с выраженным *конечным* временем жизни (такие как дескрипторы файлов, дескрипторы графического контекста, мьютексы, сокеты и т. д.). Работая с такими ресурсами как с классами, нужно постоянно быть начеку, чтобы не забыть своевременно освободить инкапсулированные ресурсы с помощью метода, вроде `close` или `dispose`. В таких случаях обычно помогает инструкция `scope` (см. раздел 3.13), но лучше, когда подобная контекстная семантика инкапсулирована в типе – раз и навсегда.

В-третьих, классы – это механизм для довольно «тяжелых» и высокоуровневых абстракций, то есть они не позволяют легко выражать «легковесные» абстракции вроде перечисляемых типов или псевдонимов для заданного типа.

D не был бы настоящим языком для системного программирования, если бы предоставлял для выражения абстракций только классы. Кроме классов в запасе у D есть структуры (типы-значения, сравнимые с классами по мощности, но с семантикой значения и без полиморфизма), типы `enum` (легковесные перечисляемые типы и простые константы), объединения (низкоуровневое хранилище с перекрыванием для разных типов) и вспомогательные механизмы определения типов, такие как `alias`. Все эти средства последовательно рассматриваются в этой главе.

7.1. Структуры

Структуры позволяют определять простые, инкапсулированные типы-значения. Удобная аналогия – тип `int`: значение типа `int` – это 4 байта, допускающие определенные операции. В `int` нет никакого скрытого состояния и никаких косвенных обращений, и две переменные типа `int` всегда ссылаются на разные значения[1]. Соглашение о структурах исключает динамический полиморфизм, переопределение методов, наследование и бесконечное время жизни. Структура – это преувеличенный тип `int`.

Как вы помните, класс ведет себя как ссылка (см. раздел 6.2), то есть вы всегда манипулируете объектом посредством ссылки на него, причем копирование ссылок лишь увеличивает количество ссылок на тот же объект без дублирования самого объекта. А структура – это тип-значение, то есть, по сути, ведет себя «как `int`»: имя жестко привязано к представленному им значению, а при копировании значения структуры на самом деле копируется целый объект, а не только ссылка.

Определяют структуру так же, как класс, за исключением следующих моментов:

[1] Не считая эквивалентных имен, создаваемых с помощью `alias`, о чем мы еще поговорим в этой главе (см. раздел 7.4).

- вместо ключевого слова `class` используется ключевое слово `struct`;
- свойственное классам наследование и реализация интерфейсов запрещены, то есть в определении структуры нельзя указать `:BaseType` или `:Interface`, и очевидно, что внутри структуры не определена ссылка `super`;
- методы структуры нельзя переопределять – все методы являются финальными (вы можете указать в определении метода структуры ключевое слово `final`, но это было бы совершенно излишне);
- нельзя применять к структуре инструкцию `synchronized` (см. главу 13);
- структуре запрещается определять конструктор по умолчанию `this()` (см. раздел 7.1.3.1);
- структуре разрешается определять конструктор копирования (postblit constructor) `this(this)` (см. раздел 7.1.3.4);
- запрещен спецификатор доступа `protected` (иначе предполагалось бы наличие структур-потомков).

Определим простую структуру:

```
struct Widget {
   // Константа
   enum fudgeFactor = 0.2;
   // Разделяемое неизменяемое значение
   static immutable defaultName = "Виджет";
   // Некоторое состояние, память под которое выделяется
   // для каждого экземпляра класса Widget
   string name = defaultName;
   uint width, height;
   // Статический метод
   static double howFudgy() {
      return fudgeFactor;
   }
   // Метод
   void changeName(string another) {
      name = another;
   }
}
```

7.1.1. Семантика копирования

Несколько заметных на глаз различий между структурами и классами есть следствие менее очевидных семантических различий. Повторим эксперимент, который мы уже проводили, обсуждая классы в разделе 6.2. На этот раз создадим структуру и объект с одинаковыми полями, а затем сравним поведение этих типов при копировании:

```
class C {
   int x = 42;
   double y = 3.14;
}
```

```
struct S {
   int x = 42;
   double y = 3.14;
}
unittest {
   C c1 = new C;
   S s1;          // Никакого оператора new для S: память выделяется в стеке
   auto c2 = c1;
   auto s2 = s1;
   c2.x = 100;
   s2.x = 100;
   assert(c1.x == 100); // c1 и c2 ссылаются на один и тот же объект...
   assert(s1.x == 42);  //   .а s2 - это настоящая копия s1
}
```

При работе со структурами нет никаких ссылок, которые можно привязывать и перепривязывать с помощью операций инициализации и присваивания. Каждое имя экземпляра структуры связано с отдельным значением. Как уже говорилось, объект-структура ведет себя *как значение*, а объект-класс – *как ссылка*. На рис. 7.1 показано положение дел сразу после определения c2 и s2.

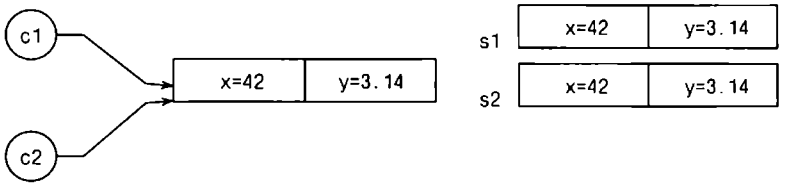

Рис. 7.1. Инструкции *auto c2 = c1;* ***для объекта-класса*** *c1* ***и*** *auto s2 = s1;* ***для объекта-структуры*** *s1* ***действуют совершенно по-разному, поскольку класс по своей природе – ссылка, а структура – значение***

В отличие от имен c1 и c2, допускающих привязку к любому объекту, имена s1 и s2 прочно привязаны к реальным объектам. Нет способа заставить два имени ссылаться на один и тот же объект-структуру (кроме ключевого слова alias, задающего простую эквивалентность имен; см. раздел 7.4), и не бывает имени структуры без закрепленного за ним значения, так что сравнение s1 is null бессмысленно и порождает ошибку во время компиляции.

7.1.2. Передача объекта-структуры в функцию

Поскольку объект типа struct ведет себя как значение, он и передается в функцию по значению.

```
struct S {
   int a, b, c;
```

```
    double x, y, z;
}
void fun(S s) {     // fun получает копию

}
```

Передать объект-структуру по ссылке можно с помощью аргумента с ключевым словом `ref` (см. раздел 5.2.1):

```
void fun(ref S s) { // fun получает ссылку

}
```

Раз уж мы заговорили о `ref`, отметим, что `this` передается по ссылке внутрь методов структуры `S` в виде скрытого параметра `ref S`.

7.1.3. Жизненный цикл объекта-структуры

В отличие от объектов-классов, объектам-структурам не свойственно бесконечное время жизни (lifetime). Время жизни для них четко ограничено – так же как для временных (стековых) объектов функций.

Чтобы создать объект-структуру, задайте имя нужного типа, как если бы вы вызывали функцию:

```
import std.math;

struct Test {
    double a = 0.4;
    double b;
}

unittest {
    // Чтобы создать объект, используйте имя структуры так,
    // как используете функцию
    auto t = Test();
    assert(t.a == 0.4 && IsNaN(t.b));
}
```

Вызов `Test()` создает объект-структуру, все поля которого инициализированы по умолчанию. В нашем случае это означает, что поле `t.a` принимает значение `0.4`, а `t.b` остается инициализированным значением `double.init`.

Вызовы `Test(1)` и `Test(1.5, 2.5)` также разрешены и инициализируют поля объекта в порядке их объявления. Продолжим предыдущий пример:

```
unittest {
    auto t1 = Test(1);
    assert(t1.a == 1 && IsNaN(t1.b));
    auto t2 = Test(1.5, 2.5);
    assert(t2.a == 1.5 && t2.b == 2.5);
}
```

Поначалу может раздражать разница в синтаксисе выражения, создающего объект-структуру Test(*‹аргументы›*), и выражения, создающего объект-класс new Test(*‹аргументы›*). D мог бы отказаться от использования ключевого слова new при создании объектов-классов, но это new напоминает программисту, что выполняется операция выделения памяти (то есть необычное действие).

7.1.3.1. Конструкторы

Конструктор структуры определяется так же, как конструктор класса (см. раздел 6.3.1):

```
struct Test {
   double a = 0.4;
   double b;
   this(double b) {
      this.b = b;
   }
}
unittest {
   auto t = Test(5);
}
```

Присутствие хотя бы одного пользовательского конструктора блокирует все упомянутые выше конструкторы, инициализирующие поля структуры:

```
auto t1 = Test(1.1, 1.2); // Ошибка!
      // Нет конструктора, соответствующего вызову Test(double, double)
```

Есть важное исключение: компилятор всегда определяет конструктор без аргументов:

```
auto t2 = Test();     // Все в порядке, создается объект
                      // с "начинкой" по умолчанию
```

Кроме того, пользовательский код не может определить собственный конструктор без аргументов:

```
struct Test {
   double a = 0.4;
   double b;
   this() { b = 0; } // Ошибка!
      // Структура не может определить конструктор по умолчанию!
}
```

Зачем нужно такое ограничение? Все из-за T.init – значения по умолчанию, определяемого каждым типом. Оно должно быть статически известно, что противоречит существованию конструктора по умолчанию, выполняющего произвольный код. (Для классов T.init – это пустая ссылка null, а не объект, построенный по умолчанию.) Правило для всех структур: конструктор по умолчанию инициализирует все поля объекта-структуры значениями по умолчанию.

7.1.3.2. Делегирование конструкторов

Скопируем пример из раздела 6.3.2 с заменой ключевого слова `class` на `struct`:

```
struct Widget {
   this(uint height) {
      this(1, height);   // Положиться на другой конструктор
   }
   this(uint width, uint height) {
      this.width = width;
      this.height = height;
   }
   uint width, height;

}
```

Код запускается, не требуя внесения каких-либо других изменений. Так же как и классы, структуры позволяют одному конструктору делегировать построение объекта другому конструктору с теми же ограничениями.

7.1.3.3. Алгоритм построения

Классу приходится заботиться о выделении динамической памяти и инициализации своего базового подобъекта (см. раздел 6.3.3). Со структурами все гораздо проще, поскольку выделение памяти – явный шаг алгоритма построения. Алгоритм построения объекта-структуры типа `T` по шагам:

1. Скопировать значение `T.init` в память, где будет размещен объект, путем копирования «сырой» памяти (а-ля `memcpy`).
2. Вызвать конструктор, если нужно.

Если инициализация некоторых или всех полей структуры выглядит как `= void`, объем работ на первом шаге можно сократить, хотя и редко намного, зато такой маневр часто порождает трудноуловимые ошибки в вашем коде (тем не менее случай оправданного применения сокращенной инициализации иллюстрирует пример с классом `Transmogrifier` в разделе 6.3.3).

7.1.3.4. Конструктор копирования this(this)

Предположим, требуется определить объект, который содержит локальный (`private`) массив и предоставляет ограниченный API для манипуляции этим массивом:

```
struct Widget {
   private int[] array;
   this(uint length) {
      array = new int[length];
   }
```

```
    int get(size_t offset) {
        return array[offset];
    }
    void set(size_t offset, int value) {
        array[offset] = value;
    }
}
```

У класса `Widget`, определенного таким образом, есть проблема: при копировании объектов типа `Widget` между копиями создается отдаленная зависимость. Судите сами:

```
unittest {
    auto w1 = Widget(10);
    auto w2 = w1;
    w1.set(5, 100);
    w2.set(5, 42);              // Также изменяет элемент w1.array[5]!
    assert(w1.get(5) == 100);   // Не проходит!?!
}
```

В чем проблема? Копирование содержимого `w1` в `w2` «поверхностно», то есть оно выполняется поле за полем, без транзитивного копирования, на какую бы память косвенно ни ссылалось каждое из полей. При копировании массива память под новый массив не выделяется; копируются лишь границы массива (см. раздел 4.1.4). После копирования `w1` и `w2` действительно обладают различными полями с массивами, но ссылаются эти поля на одну и ту же область памяти. Такой объект, являющийся значением, но содержащий неявные разделяемые ссылки, можно в шутку назвать «клуктурой», то есть гибридом структуры (семантика значения) и класса (семантика ссылки)[1].

Обычно требуется, чтобы структура действительно вела себя как значение, то есть чтобы копия становилась полностью независимой от своего источника. Для этого определите конструктор копирования так:

```
struct Widget {
    private int[] array;
    this(uint length) {
        array = new int[length];
    }
    // Конструктор копирования
    this(this) {
        array = array.dup;
    }
    // Как раньше
    int get(size_t offset) { return array[offset]; }
    void set(size_t offset, int value) { array[offset] = value; }
}
```

[1] Термин «клуктура» предложил Бартош Милевски.

Конструктор копирования вступает в силу во время копирования объекта. Чтобы инициализировать объект приемник с помощью объекта источник того же типа, компилятор должен выполнить следующие шаги:

1. Скопировать участок «сырой» памяти объекта источник в участок «сырой» памяти объекта приемник.
2. Транзитивно для каждого поля, содержащего другие поля (то есть поля, содержащего другое поле, содержащее третье поле, ...), для которого определен метод this(this), вызвать эти конструкторы снизу вверх (начиная от наиболее глубоко вложенного поля).
3. Вызвать метод this(this) с объектом приемник.

Оригинальное название конструктора копирования «postblit constructor» происходит от «blit» — популярной аббревиатуры понятия «block transfer», означавшего копирование «сырой» памяти. Язык применяет «сырое» копирование при инициализации и разрешает сразу после этого воспользоваться ловушкой. В предыдущем примере конструктор копирования превращает только что полученный псевдоним массива в настоящую, полномасштабную копию, гарантируя, что с этого момента между объектом-оригиналом и объектом-копией не будет ничего общего. Теперь, после добавления конструктора копирования, модуль легко проходит этот тест:

```
unittest {
   auto w1 = Widget(10);
   auto w2 = w1;              // this(this) здесь вызывается с w2
   w1.set(5, 100);
   w2.set(5, 42);
   assert(w1.get(5) == 100); // Пройдено
}
```

Вызов конструктора копирования вставляется в каждом случае копирования какого-либо объекта при явном или неявном создании новой переменной. Например, при передаче объекта типа Widget по значению в функцию также создается копия:

```
void fun(Widget w) {          // Передать по значению
   w.set(2, 42);
}

void gun(ref Widget w) {      // Передать по ссылке
   w.set(2, 42);
}

unittest {
   auto w1 = Widget(10);
   w1.set(2, 100);
   fun(w1);                   // Здесь создается копия
   assert(w1.get(2) == 100); // Тест пройден
   gun(w1);                   // А здесь копирования нет
```

```
    assert(w1.get(2) == 42);  // Тест пройден
}
```

Второй шаг (часть с «транзитивным полем») процесса конструирования при копировании заслуживает особого внимания. Основанием для такого поведения является *инкапсуляция*: конструктор копирования объекта-структуры должен быть вызван даже тогда, когда эта структура встроена в другую. Предположим, например, что мы решили сделать `Widget` членом другой структуры, которая в свою очередь является членом третьей структуры:

```
struct Widget2 {
    Widget w1;
    int x;
}

struct Widget3 {
    Widget2 w2;
    string name;
    this(this) {
        name = name ~ " (copy)";
    }
}
```

Теперь, если потребуется копировать объекты, содержащие другие объекты типа `Widget`, будет очень некстати, если компилятор забудет, как нужно копировать подобъекты типа `Widget`. Вот почему при копировании объектов типа `Widget2` инициируется вызов конструктора `this(this)` для подобъекта `w1`, невзирая на то, что `Widget2` вообще об этом ничего не знает. Кроме того, при копировании объектов типа `Widget3` конструктор `this(this)` по-прежнему вызывается применительно к полю `w1` поля `w2`. Внесем ясность:

```
unittest {
    Widget2 a;
    a.w1 = Widget(10);          // Выделить память под данные
    auto b = a;                 // this(this) вызывается для b.w1
    assert(a.w1.array !is b.w1.array);        // Тест пройден
    Widget3 c;
    c.w2.w1 = Widget(20);
    auto d = c;                 // this(this) вызывается для d.w2.w1
    assert(c.w2.w1.array !is d.w2.w1.array); // Тест пройден
}
```

Вкратце, если вы определите для некоторой структуры конструктор копирования `this(this)`, компилятор позаботится о том, чтобы конструктор копирования вызывался в каждом случае копирования этого объекта-структуры независимо от того, является ли он самостоятельным объектом или частью более крупного объекта-структуры.

7.1.3.5. Аргументы в пользу this(this)

Зачем был введен конструктор копирования? Ведь ничего подобного в других языках пока нет. Почему бы просто не передавать исходный объект в будущую копию (как это делает C++)?

```
// Это не D
struct S {
   this(S another) {     }
// Или
   this(ref S another) {     }
}
```

Опыт с C++ показал, что основная причина неэффективности программ на C++ – злоупотребление копированием объектов. Чтобы сократить потери эффективности по этой причине, C++ устанавливает ряд случаев, в которых компилятор может пропускать вызов конструктора копирования (copy elision). Правила для этих случаев очень быстро усложнились, но все равно не охватывали все моменты, когда можно обойтись без конструирования, то есть проблема осталась не решенной. Развивающийся стандарт C++ затрагивает эти вопросы, определяя новый тип «ссылка на r-значение», позволяющий пользователю управлять пропусками вызова конструктора копирования, но плата за это – еще большее усложнение языка.

Благодаря конструктору копирования подход D становится простым и во многом автоматизируемым. Начнем с того, что объекты в D должны быть *перемещаемыми*, то есть не должны зависеть от своего расположения: копирование «сырой» памяти позволяет переместить объект в другую область памяти, не нарушая его целостность. Тем не менее это ограничение означает, что объект не может содержать так называемые *внутренние указатели* – адреса подобъектов, являющихся его частями. Без этой техники можно обойтись, так что D попросту ее исключает. Создавать объекты с внутренними указателями в D запрещается, и компилятор, как и подсистема времени исполнения, вправе предполагать, что это правило соблюдается. Перемещаемые объекты открывают для компилятора и подсистемы времени исполнения (например, для сборщика мусора) большие возможности, позволяющие программам стать более быстрыми и компактными.

Благодаря перемещаемости объектов копирование объектов становится логическим продолжением перемещения объектов: конструктор копирования this(this) делает копирование объектов эквивалентом перемещения с возможной последующей пользовательской обработкой. Таким образом, пользовательский код не может изменить поля исходного объекта (что очень хорошо, поскольку копирование не должно затрагивать объект-источник), но зато может корректировать поля, которые не должны неявно разделять состояние с объектом-источником. Чтобы избежать лишнего копирования, компилятор вправе по собственному усмотрению не вставлять вызов this(this), если может доказать, что источник

копии не будет использован после завершения процесса копирования. Рассмотрим, например, функцию, возвращающую объект типа Widget (определенный выше) по значению:

```
Widget hun(uint x) {
   return Widget(x * 2);
}

unittest {
   auto w = hun(1000);
   ...
}
```

Наивный подход: просто создать объект типа Widget внутри функции hun, а затем скопировать его в переменную w, применив побитовое копирование с последующим вызовом this(this). Но это было бы слишком расточительно: D полагается на перемещаемость объектов, так почему бы попросту не переместить в переменную w уже отживший свое временный объект, созданный функцией hun? Разницу никто не заметит, поскольку после того, как функция hun вернет результат, временный объект уже не нужен. Если в лесу упало дерево и никто этого не слышит, то легче переместить его, чем копировать. Похожий (но не идентичный) случай:

```
Widget iun(uint x) {
   auto result = Widget(x * 2);

   return result;
}

unittest {
   auto w = iun(1000);

}
```

В этом случае переменная result тоже уходит в небытие сразу же после того, как iun вернет управление, поэтому в вызове this(this) необходимости нет. Наконец, еще более тонкий случай:

```
void jun(Widget w) {

}

unittest {
   auto w = Widget(1000);
   ... // ‹код1›
   jun(w);
   ... // ‹код2›
}
```

В этом случае сложнее выяснить, можно ли избавиться от вызова this(this). Вполне вероятно, что *‹код2›* продолжает использовать w, и то-

гда перемещение этого значения из unittest в jun было бы некорректным[1].

Ввиду всех перечисленных соображений в D приняты следующие правила пропуска вызова конструктора копирования:

- Все анонимные r-значения перемещаются, а не копируются. Вызов конструктора копирования this(this) всегда пропускается, если оригиналом является анонимное r-значение (то есть временный объект, как в функции hun выше).
- В случае именованных временных объектов, которые создаются внутри функции и располагаются в стеке, а затем возвращаются этой функцией в качестве результата, вызов конструктора копирования this(this) пропускается.
- Нет никаких гарантий, что компилятор воспользуется другими возможностями пропустить вызов конструктора копирования.

Но иногда требуется предписать компилятору выполнить перемещение. Фактически это выполняет функция move из модуля std.algorithm стандартной библиотеки:

```
import std.algorithm;

void kun(Widget w) {

}

unittest {
   auto w = Widget(1000);
   ... // ‹код1›
   // Вставлен вызов move
   kun(move(w));
   assert(w == Widget.init); // Пройдено
   ... // ‹код2›
}
```

Вызов функции move гарантирует, что w будет перемещена, а ее содержимое будет заменено пустым, сконструированным по умолчанию объектом типа Widget. Кстати, это один из тех случаев, где пригодится неизменяемый и не порождающий исключения конструктор по умолчанию Widget.init (см. раздел 7.1.3.1). Без него сложно было бы найти способ оставить источник перемещения в строго определенном пустом состоянии.

7.1.3.6. Уничтожение объекта и освобождение памяти

Структура может определять деструктор с именем ~this():

```
import std.stdio;

struct S {
```

[1] Кроме того, ‹код1› может сохранить указатель на значение w, которое использует ‹код2›.

```
    int x = 42;
    ~this() {
        writeln("Структура S с содержимым ", x, " исчезает. Пока!");
    }
}

void main() {
    writeln("Создание объекта типа S.");
    {
        S object;
        writeln("Внутри области видимости объекта ");
    }
    writeln("Вне области видимости объекта");
}
```

Эта программа гарантированно выведет на экран:

```
Создание объекта типа S.
Внутри области видимости объекта
Структура S с содержимым 42 исчезает. Пока!
Вне области видимости объекта.
```

Каждая структура обладает *временем жизни в пределах области видимости* (*scoped lifetime*), то есть ее жизнь действительно заканчивается с окончанием области видимости объекта. Подробнее:

- **время жизни нестатического объекта, определенного внутри функции, заканчивается в конце текущей области видимости (то есть контекста) до уничтожения всех объектов-структур, определенных перед ним;**
- **время жизни объекта, определенного в качестве члена другой структуры, заканчивается непосредственно после окончания времени жизни включающего объекта;**
- **время жизни объекта, определенного в контексте модуля, бесконечно; если вам нужно вызвать деструктор этого объекта, сделайте это в деструкторе модуля (см. раздел 11.3);**
- **время жизни объекта, определенного в качестве члена класса, заканчивается в тот момент, когда сборщик мусора забирает память включающего объекта.**

Язык гарантирует автоматический вызов деструктора `~this` по окончании времени жизни объекта-структуры, что очень удобно, если вы хотите автоматически выполнять такие операции, как закрытие файлов и освобождение всех важных ресурсов.

Оригинал копии, использующей конструктор копирования, подчиняется обычным правилам для времени жизни, но деструктор оригинала копии, полученной перемещением «сырой» памяти без вызова `this(this)`, не вызывается.

Освобождение памяти объекта-структуры по идее выполняется сразу же после деструкции.

7.1.3.7. Алгоритм уничтожения структуры

По умолчанию объекты-структуры уничтожаются в порядке, строго обратном порядку их создания. То есть первым уничтожается объект-структура, определенный в заданной области видимости последним:

```
import std.conv, std.stdio;

struct S {
   private string name;
   this(string name) {
      writeln(name, " создан.");
      this.name = name;
   }
   ~this() {
      writeln(name, " уничтожен.");
   }
}

void main() {
   auto obj1 = S("первый объект");
   foreach (i; 0    3) {
      auto obj = S(text("объект "  i));
   }
   auto obj2 = S("последний объект");
}
```

Эта программа выведет на экран:

```
первый объект создан.
объект 0 создан.
объект 0 уничтожен.
объект 1 создан.
объект 1 уничтожен.
объект 2 создан.
объект 2 уничтожен.
последний объект создан.
последний объект уничтожен.
первый объект уничтожен.
```

Как и ожидалось, объект, созданный первым, был уничтожен последним. На каждой итерации цикл входит в контекст и выходит из контекста управляемой инструкции.

Можно явно инициировать вызов деструктора объекта-структуры с помощью инструкции `clear(объект);`. С функцией `clear` мы уже познакомились в разделе 6.3.5. Тогда она оказалась полезной для уничтожения состояния объекта-класса. Для объектов-структур функция `clear` делает то же самое: вызывает деструктор, а затем копирует биты значения `.init` в область памяти объекта. В результате получается правильно сконструированный объект, правда, без какого-либо интересного содержания.

7.1.4. Статические конструкторы и деструкторы

Структура может определять любое число статических конструкторов и деструкторов. Это средство полностью идентично одноименному средству для классов, с которым мы уже встречались в разделе 6.3.6.

```
import std.stdio;

struct A {
   static ~this() {
      writeln("Первый статический деструктор");
   }
   ..
   static this() {
      writeln("Первый статический конструктор ");
   }

   static this() {
      writeln("Второй статический конструктор");
   }

   static ~this() {
      writeln("Второй статический деструктор");
   }
}

void main() {
   writeln("Внимание, говорит main");
}
```

Парность статических конструкторов и деструкторов не требуется. Подсистема поддержки времени исполнения не делает ничего интересного – просто выполняет все статические конструкторы перед вычислением функции `main` в порядке их определения. По завершении выполнения `main` подсистема поддержки времени исполнения так же скучно вызывает все статические деструкторы в порядке, обратном порядку их определения. Предыдущая программа выведет на экран:

```
Первый статический конструктор
Второй статический конструктор
Внимание, говорит main
Второй статический деструктор
Первый статический деструктор
```

Порядок выполнения очевиден для статических конструкторов и деструкторов, расположенных внутри одного модуля, но в случае нескольких модулей не всегда все так же ясно. Порядок выполнения статических конструкторов и деструкторов из разных модулей определен в разделе 6.3.6.

7.1.5. Методы

Структуры могут определять функции-члены, также называемые методами. Поскольку в случае структур о наследовании и переопределении речи нет, методы структур лишь немногим больше, чем функции.

Нестатические методы структуры S принимают скрытый параметр this по ссылке (эквивалент параметра ref S). Поиск имен внутри методов структуры производится так же, как и внутри методов класса: параметры перекрывают одноименные внутренние элементы структуры, а имена внутренних элементов структуры перекрывают те же имена, объявленные на уровне модуля.

```
void fun(int x) {
   assert(x != 0);
}

// Проиллюстрируем правила поиска имен
struct S {
   int x = 1;
   static int y = 324;

   void fun(int x) {
      assert(x == 0);       // Обратиться к параметру x
      assert(this.x == 1); // Обратиться к внутреннему элементу x
   }

   void gun() {
      fun(0);               // Вызвать метод fun
      .fun(1);              // Вызвать функцию fun,
                            // определенную на уровне модуля
   }

   // Тесты модуля могут быть внутренними элементами структуры
   unittest {
      S obj;
      obj.gun();
      assert(y == 324);     // Тесты модуля, являющиеся
                            // "внутренними элементами"
                            // видят статические данные
   }
}
```

Кроме того, в этом примере есть тест модуля, определенный внутри структуры. Такие тесты модуля, являющиеся «внутренними элементами», не наделены никаким особым статусом, но их очень удобно вставлять после каждого определения метода. Коду тела внутреннего теста модуля доступна та же область видимости, что и обычным статическим методам: например, тесту модуля в предыдущем примере не требуется снабжать статическое поле y префиксом S, как это не потребовалось бы любому методу структуры.

Некоторые особые методы заслуживают более тщательного рассмотрения. К ним относятся оператор присваивания opAssign, используемый оператором =, оператор равенства opEquals, используемый операторами == и !=, а также упорядочивающий оператор opCmp, используемый операторами <, <=, >= и >. На самом деле, эта тема относится к главе 12, так как затрагивает вопрос перегрузки операторов, но эти операторы особенные: компилятор может сгенерировать их автоматически, со всем их особым поведением.

7.1.5.1. Оператор присваивания

По умолчанию, если задать:

```
struct Widget {      } // Определен так же, как в разделе 7.1.3.4
Widget w1, w2;

w1 = w2;
```

то присваивание делается через копирование всех внутренних элементов по очереди. В случае типа Widget такой подход может вызвать проблемы, о которых говорилось в разделе 7.1.3.4. Если помните, структура Widget обладает внутренним локальным массивом типа int[], и планировалось, что он будет индивидуальным для каждого объекта типа Widget. В ходе последовательного присваивания полей объекта w2 объекту w1 поле w2.array будет присвоено полю w1.array, но это будет только простое присваивание границ массива — в действительности, содержимое массива скопировано не будет. Этот момент необходимо подкорректировать, поскольку на самом деле мы хотим создать *дубликат* массива оригинальной структуры и присвоить его целевой структуре.

Пользовательский код может перехватить присваивание, определив метод opAssign. По сути, если lhs определяет opAssign с совместимой сигнатурой, присваивание lhs = rhs транслируется в lhs.opAssign(rhs), иначе если lhs и rhs имеют один и тот же тип, выполняется обычное присваивание поле за полем. Давайте определим метод Widget.opAssign:

```
struct Widget {
   private int[] array;
   ..  // this(uint), this(this), и т.д.
   ref Widget opAssign(ref Widget rhs) {
      array = rhs.array.dup;
      return this;
   }
}
```

Оператор присваивания возвращает ссылку на this, тем самым позволяя создавать цепочки присваиваний а-ля w1 = w2 = w3, которые компилятор заменяет на w1.opAssign(w2.opAssign(w3)).

Осталась одна проблема. Рассмотрим присваивание:

```
Widget w;

w = Widget(50); // Ошибка!
   // Невозможно привязать r-значение типа Widget к ссылке ref Widget!
```

Проблема в том, что метод opAssign в таком виде, в каком он определен сейчас, ожидает аргумент типа ref Widget, то есть l-значение типа Widget. Чтобы помимо l-значений можно было бы присваивать еще и r-значения, структура Widget должна определять *два* оператора присваивания:

```
import std.algorithm;

struct Widget {
   private int[] array;
       // this(uint), this(this), и т. д.
   ref Widget opAssign(ref Widget rhs) {
      array = rhs.array.dup;
      return this;
   }
   ref Widget opAssign(Widget rhs) {
      swap(array, rhs.array);
      return this;
   }
}
```

В версии метода, принимающей r-значения, уже отсутствует обращение к свойству .dup. Почему? Ну, r-значение (а с ним и его массив) – это практически собственность второго метода opAssign: оно было скопировано перед входом в функцию и будет уничтожено сразу же после того, как функция вернет управление. Это означает, что больше нет нужды дублировать rhs.array, потому что его потерю никто не ощутит. Достаточно лишь поменять местами rhs.array и this.array. Функция opAssign возвращает результат, и rhs и старый массив объекта this уходят в никуда, а this остается с массивом, ранее принадлежавшим rhs, – совершенное сохранение состояния.

Теперь можно совсем убрать первую перегруженную версию оператора opAssign: та версия, что принимает rhs по значению, заботится обо всем сама (l-значения автоматически конвертируются в r-значения). Но оставив версию с l-значением, мы сохраняем точку, через которую можно оптимизировать работу оператора присваивания. Вместо того чтобы дублировать структуру-оригинал с помощью свойства .dup, метод opAssign может проверять, достаточно ли в текущем массиве места для размещения нового содержимого, и если да, то достаточно и записи поверх старого массива на том же месте.

```
// Внутри Widget
ref Widget opAssign(ref Widget rhs) {
   if (array.length < rhs.array.length) {
      array = rhs.array.dup;
   } else {
```

```
        // Отрегулировать длину
        array.length = rhs.array.length;
        // Скопировать содержимое массива array (см. раздел 4.1.7)
        array[] = rhs.array[];
    }
    return this;
}
```

7.1.5.2. Сравнение структур на равенство

Средство для сравнения объектов-структур предоставляется «в комплекте» – это операторы == и !=. Сравнение представляет собой поочередное сравнение внутренних элементов объектов и возвращает false, если хотя бы два соответствующих друг другу элемента сравниваемых объектов не равны, иначе результатом сравнения является true.

```
struct Point {
    int x, y;
}
unittest {
    Point a, b;
    assert(a == b);
    a.x = 1;
    assert(a != b);
}
```

Чтобы определить собственный порядок сравнения, определите метод opEquals:

```
import std.math, std.stdio;

struct Point {
    float x = 0, y = 0;
    // Добавлено
    bool opEquals(ref const Point rhs) const {
        // Выполнить приблизительное сравнение
        return approxEqual(x, rhs.x) && approxEqual(y, rhs.y);
    }
}

unittest {
    Point a, b;
    assert(a == b);
    a.x = 1e-8;
    assert(a == b);
    a.y = 1e-1;
    assert(a != b);
}
```

По сравнению с методом opEquals для классов (см. раздел 6.8.3) метод opEquals для структур гораздо проще: ему не нужно беспокоиться о корректности своих действий из-за наследования. Компилятор попросту

заменяет сравнение объектов-структур на вызов метода opEquals. Конечно, применительно к структурам остается требование определять осмысленный метод opEquals: рефлексивный, симметричный и транзитивный. Заметим, что хотя метод Point.opEquals выглядит довольно осмысленно, он не проходит тест на транзитивность. Лучшим вариантом оператора сравнения на равенство было бы сравнение двух объектов типа Point, значения координат которых предварительно усечены до своих старших разрядов. Такую проверку было бы гораздо проще сделать транзитивной.

Если структура содержит внутренние элементы, определяющие методы opEquals, а сама такой метод не определяет, при сравнении все равно будут вызваны существующие методы opEquals внутренних элементов. Продолжим работать с примером, содержащим структуру Point:

```
struct Rectangle {
   Point leftBottom, rightTop;
}

unittest {
   Rectangle a, b;
   assert(a == b);
   a.leftBottom.x = 1e-8;
   assert(a == b);
   a.rightTop.y = 5;
   assert(a != b);
}
```

Для любых двух объектов a и b типа Rectangle вычисление a == b эквивалентно вычислению выражения

```
a.leftBottom == b.leftBottom && a.rightTop == b.rightTop
```

что в свою очередь можно переписать так:

```
a.leftBottom.opEquals(b.leftBottom) &&
   a.rightTop.opEquals(b.rightTop)
```

Этот пример также показывает, что сравнение выполняется в порядке объявления полей (т. е. поле leftBottom проверяется до проверки rightTop), и если встретились два неравных поля, сравнение завершается до того, как будут проверены все поля, благодаря сокращенному вычислению логических связок, построенных с помощью оператора && (short circuit evaluation).

7.1.6. Статические внутренние элементы

Структура может определять статические данные и статические внутренние функции. Помимо ограниченной видимости и подчинения правилам доступа (см. раздел 7.1.7) режим работы статических внутренних функций ничем не отличается от режима работы обычных функций.

Нет скрытого параметра this, не вовлечены никакие другие особые механизмы.

Точно так же статические данные схожи с глобальными данными, определенными на уровне модуля (см. раздел 5.2.4), во всем, кроме видимости и ограничений доступа, наложенных на эти статические данные родительской структурой.

```
import std.stdio;

struct Point {
   private int x, y;
   private static string formatSpec = "(%s %s)\n";
   static void setFormatSpec(string newSpec) {
          // Проверить корректность спецификации формата
      formatSpec = newSpec;
   }

   void print() {
      writef(formatSpec, x, y);
   }
}

void main() {
   auto pt1 = Point(1, 2);
   pt1.print();
   // Вызвать статическую внутреннюю функцию,
   // указывая ее принадлежность префиксом Point или pt1
   Point.setFormatSpec("[%s, %s]\n");
   auto pt2 = Point(5, 3);
   // Новая спецификация действует на все объекты типа Point
   pt1.print();
   pt2.print();
}
```

Эта программа выведет на экран:

```
(1 2)
[1, 2]
[5, 3]
```

7.1.7. Спецификаторы доступа

Структуры подчиняются спецификаторам доступа private (см. раздел 6.7.1), package (см. раздел 6.7.2), public (см. раздел 6.7.4) и export (см. раздел 6.7.5) тем же образом, что и классы. Спецификатор protected применительно к структурам не имеет смысла, поскольку структуры не поддерживают наследование.

За подробной информацией обратитесь к соответствующим разделам. А здесь мы лишь вкратце напомним смысл спецификаторов:

```
struct S {
   private int a; // Доступен в пределах текущего файла и в методах S
   package int b; // Доступен в пределах каталога текущего файла
   public int c;  // Доступен в пределах текущего приложения
   export int d;  // Доступен вне текущего приложения
                  // (там, где оно используется)
}
```

Заметим, что хотя ключевое слово `export` разрешено везде, где синтаксис допускает применение спецификатора доступа, семантика этого ключевого слова зависит от реализации.

7.1.8. Вложенность структур и классов

Часто бывает удобно вложить в структуру другую структуру или класс. Например, контейнер дерева можно представить как оболочку-структуру с простым интерфейсом поиска, а внутри нее для определения узлов дерева использовать полиморфизм.

```
struct Tree {
private:
   class Node {
      int value;
      abstract Node left();
      abstract Node right();
   }
   class NonLeaf : Node {
      Node _left, _right;
      override Node left() { return _left; }
      override Node right() { return _right; }
   }
   class Leaf : Node {
      override Node left() { return null; }
      override Node right() { return null; }
   }
   // Данные
   Node root;
public:
   void add(int value) {     }
   bool search(int value) {     }
}
```

Аналогично структура может быть вложена в другую структуру...

```
struct Widget {
private:
   struct Field {
      string name;
      uint x, y;
   }
   Field[] fields;
```

```
public:

}
```

...и наконец, структура может быть вложена в класс.

```
class Window {
   struct Info {
      string name;
      Window parent;
      Window[] children;
   }
   Info getInfo();
   ..
}
```

В отличие от классов, вложенных в другие классы, вложенные структуры и классы, вложенные в другие структуры, не обладают никаким скрытым внутренним элементом `outer` – никакой специальный код не генерируется. Такие вложенные типы определяются в основном со структурной целью – чтобы получить нужное управление доступом.

7.1.9. Структуры, вложенные в функции

Вспомним, что говорилось в разделе 6.11.1: вложенные классы находятся в привилегированном положении, ведь они обладают особыми, уникальными свойствами. Вложенному классу доступны параметры и локальные переменные включающей функции. Если вы возвращаете вложенный класс в качестве результата функции, компилятор даже размещает кадр функции в динамической памяти, чтобы параметры и локальные переменные функции выжили после того, как она вернет управление.

Для единообразия и согласованности D оказывает структурам, вложенным в функции, те же услуги, что и классам, вложенным в функции. Вложенная структура может обращаться к параметрам и локальным переменным включающей функции:

```
void fun(int a) {
   int b;
   struct Local {
      int c;
      int sum() {
         // Обратиться к параметру, переменной
         // и собственному внутреннему элементу структуры Local
         return a + b + c;
      }
   }
   Local obj;
   int x = obj.sum();
   // (void*).sizeof - размер указателя на окружение
   // int.sizeof - размер единственного поля структуры
```

```
    assert(Local.sizeof == (void*).sizeof + int.sizeof);
}
unittest {
    fun(5);
}
```

Во вложенные структуры встраивается волшебный «указатель на кадр», с помощью которого они получают доступ к внешним значениям, таким как a и b в этом примере. Из-за этого дополнительного состояния размер объекта Local не 4 байта, как можно было ожидать, а 8 (на 32-разрядной машине) – еще 4 байта занимает указатель на кадр. Если хотите определить вложенную структуру без этого багажа, просто добавьте в определение структуры Local ключевое слово static перед ключевым словом struct – тем самым вы превратите Local в обычную структуру, то есть закроете для нее доступ к a и b.

Вложенные структуры практически бесполезны, разве что, по сравнению со вложенными классами, позволяют избежать беспричинного ограничения. Функции не могут возвращать объекты вложенных структур, так как вызывающему их коду недоступна информация о типах таких объектов. Используя замысловатые вложенные структуры, код неявно побуждает создавать все больше сложных функций, а в идеале именно этого надо избегать в первую очередь.

7.1.10. Порождение подтипов в случае структур. Атрибут @disable

К структурам неприменимы наследование и полиморфизм, но этот тип данных по-прежнему поддерживает конструкцию alias this, впервые представленную в разделе 6.13. С помощью alias this можно сделать структуру подтипом любого другого типа. Определим, к примеру, простой тип Final, поведением очень напоминающий ссылку на класс – во всем, кроме того что переменную типа Final невозможно переприв язать! Пример использования переменной Final:

```
import std.stdio;

class Widget {
    void print() {
        writeln("Привет, я объект класса Widget. Вот, пожалуй, и все обо мне.");
    }
}

unittest {
    auto a = Final!Widget(new Widget);
    a.print();      // Все в порядке, просто печатаем a
    auto b = a;     // Все в порядке, a и b привязаны
                    // к одному и тому же объекту типа Widget
    a = b;          // Ошибка!
                    // opAssign(Final!Widget) деактивизирован!
```

```
    a = new Widget; // Ошибка!
                    // Невозможно присвоить значение r-значению,
                    // возвращенному функцией get()!
}
```

Предназначение типа Final – быть особым видом ссылки на класс, раз и навсегда привязанной к одному объекту. Такие «преданные» ссылки полезны для реализации множества проектных идей.

Первый шаг – избавиться от присваивания. Проблема в том, что оператор присваивания генерируется автоматически, если не объявлен пользователем, поэтому структура Final должна вежливо указать компилятору не делать этого. Для этого предназначен атрибут @disable:

```
struct Final(T) {
    // Запретить присваивание
    @disable void opAssign(Final);

}
```

С помощью атрибута @disable можно запретить и другие сгенерированные функции, например сравнение.

До сих пор все шло хорошо. Чтобы реализовать Final!T, нужно с помощью конструкции alias this сделать Final(T) подтипом T, но чтобы при этом полученный тип не являлся l-значением. Ошибочное решение выглядит так:

```
// Ошибочное решение
struct Final(T) {
    private T payload;
    this(T bindTo) {
        payload = bindTo;
    }
    // Запретить присваивание
    @disable void opAssign(Final);
    // Сделать Final(T) подклассом T
    alias payload this;
}
```

Структура Final хранит ссылку на себя в поле payload, которое инициализируется в конструкторе. Кроме того, объявив, но не определяя метод opAssign, структура эффективно «замораживает» присваивание. Таким образом, клиентский код, пытающийся присвоить значение объекту типа Final!T, или не сможет обратиться к payload (из-за private), или получит ошибку во время компоновки.

Ошибка Final – в использовании инструкции alias payload this;. Этот тест модуля делает что-то непредусмотренное:

```
class A {
    int value = 42;
    this(int x) { value = x; }
}
```

```
unittest {
   auto v = Final!A(new A(42));
   void sneaky(ref A ra) {
      ra = new A(4242);
   }
   sneaky(v); // Хм-м-м..
   assert(v.value == 4242); // Проходит?!?
}
```

alias payload this действует довольно просто: каждый раз, когда значение объект типа Final!T используется в недопустимом для этого типа контексте, компилятор вместо объект пишет объект.payload (то есть делает объект.payload *псевдонимом* для объекта в соответствии с именем и синтаксисом конструкции alias). Но выражение объект.payload представляет собой непосредственное обращение к полю объект, следовательно, является l-значением. Это l-значение привязано к переданному по ссылке параметру функции sneaky и, таким образом, позволяет sneaky напрямую изменять значение поля объекта v.

Чтобы это исправить, нужно сделать объект псевдонимом r-значения. Так мы получим полную функциональность, но ссылка, сохраненная в payload, станет неприкосновенной. Очень просто осуществить привязку к r-значению с помощью свойства (объявленного с атрибутом @property), возвращающего payload по значению:

```
struct Final(T) {
   private T payload;
   this(T bindTo) {
      payload = bindTo;
   }
   // Запретить присваивание, оставив метод opAssign неопределенным
   private void opAssign(Final);
   // Сделать Final(T) подклассом T,
   // не разрешив при этом перепривязывать payload
   @property T get() { return payload; }
   alias get this;
}
```

Ключевой момент в новом определении структуры — то, что метод get возвращает значение типа T, а не ref T. Конечно, объект, на который ссылается payload, изменить можно (если хотите избежать этого, ознакомьтесь с квалификаторами const и immutable; см. главу 8). Но структура Final свои обязательства теперь выполняет. Во-первых, для любого типа класса T справедливо, что Final!T ведет себя как T. Во-вторых, однажды привязав переменную типа Final!T к некоторому объекту с помощью конструктора, вы не сможете ее перепривязать ни к какому другому объекту. В частности, тест модуля, из-за которого пришлось отказаться от предыдущего определения Final, больше не компилируется, поскольку вызов sneaky(v) теперь некорректен: r-значение типа A (неявно полученное из v с помощью v.get) не может быть привязано к ref A, как требуется функции sneaky для ее черных дел.

В нашей бочке меда осталась только одна ложка дегтя (на самом деле, всего лишь чайная ложечка), от которой надо избавиться. Всякий раз, когда тип, подобный Final, использует конструкцию alias get this, необходимо уделять особое внимание собственным идентификаторам Final, перекрывающим одноименные идентификаторы, определенные в типе, псевдонимом которого становится Final. Предположим, мы используем тип Final!Widget, а класс Widget и сам определяет свойство get:

```
class Widget {
   private int x;
   @property int get() { return x; }
}
unittest {
   auto w = Final!Widget(new Widget);
   auto x = w.get; // Получает Widget из Final, а не int из Widget
}
```

Чтобы избежать таких коллизий, воспользуемся соглашением об именовании. Для надежности будем просто добавлять к именам видимых свойств имя соответствующего типа:

```
struct Final(T) {
   private T Final_payload;
   this(T bindTo) {
      Final_payload = bindTo;
   }
   // Запретить присваивание
   @disable void opAssign(Final);
   // Сделать Final(T) подтипом T,
   // не разрешив при этом перепривязывать payload
   @property T Final_get() { return Final_payload; }
   alias Final_get this;
}
```

Соблюдение такого соглашения сводит к минимуму риск непредвиденных коллизий. (Конечно, иногда можно намеренно перехватывать некоторые методы, оставив вызовы к ним за перехватчиком.)

7.1.11. Взаимное расположение полей. Выравнивание

Как располагаются поля в объекте-структуре? D очень консервативен в отношении структур: он располагает элементы их содержимого в том же порядке, в каком они указаны в определении структуры, но сохраняет за собой право вставлять между полями *отступы* (*padding*). Рассмотрим пример:

```
struct A {
   char a;
   int b;
   char c;
}
```

Если бы компилятор располагал поля в точном соответствии с размерами, указанными в структуре A, то адресом поля b оказался бы адрес объекта A плюс 1 (поскольку поле a типа char занимает ровно 1 байт). Но такое расположение проблематично, ведь современные компьютерные системы извлекают данные только блоками по 4 или 8 байт, то есть могут извлекать только данные, расположенные по адресам, кратным 4 и 8 соответственно. Предположим, объект типа A расположен по «хорошему» адресу, например кратному 8. Тогда адрес поля b точно окажется не в лучшем районе города. Чтобы извлечь b, процессору придется повозиться, ведь нужно будет «склеивать» значение b, собирая его из кусочков размером в байт. Усугубляет ситуацию то, что в зависимости от компилятора и низкоуровневой архитектуры аппаратного обеспечения эта операция сборки может быть выполнена лишь в ответ на прерывание ядра «обращение к невыровненным данным», обработка которого требует своих (и немалых) накладных расходов [28]. А это вам не семечки щелкать: такая дополнительная гимнастика легко снижает скорость доступа на несколько порядков.

Вот почему современные компиляторы располагают данные в памяти с *отступами*. Компилятор вставляет в объект дополнительные байты, чтобы обеспечить расположение всех полей с удобными смещениями. Таким образом, выделение под объекты областей памяти с адресами, кратными слову, гарантирует быстрый доступ ко всем внутренним элементам этих объектов. На рис. 7.2 показано расположение полей типа A по схеме с отступами.

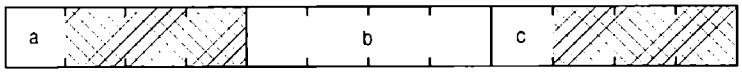

***Рис. 7.2.** Расположение полей типа A по схеме с отступами. Заштрихованные области – это отступы, вставленные для правильного выравнивания. Компилятор вставляет в объект две лакуны, тем самым добавляя 6 байт простаивающего места или 50% общего размера объекта*

Полученное расположение полей характеризуется обилием отступов (заштрихованных областей). В случае классов компилятор волен упорядочивать поля по собственному усмотрению, но при работе со структурой есть смысл позаботиться о расположении данных, если объем используемой памяти имеет значение. Лучше всего расположить поле типа int первым, а после него – два поля типа char. При таком порядке полей структура займет 64 бита, включая 2 байта отступа.

Каждое из полей объекта обладает известным во время компиляции смещением относительно начального адреса объекта. Это смещение всегда одинаково для всех объектов заданного типа в рамках одной программы (оно может меняться от компиляции к компиляции, но не от запуска к запуску). Смещение доступно пользовательскому коду как значение

свойства .offsetof, неявно определенного для каждого поля класса или структуры:

```
import std.stdio;

struct A {
   char a;
   int b;
   char c;
}

void main() {
   A x;
   writefln("%s %s %s"  x.a.offsetof, x.b.offsetof, x.c.offsetof);
}
```

Эталонная реализация компилятора выведет 0 4 8, открывая схему расположения полей, которую мы уже видели на рис. 7.2. Не совсем удобно, что для доступа к некоторой статической информации о типе A приходится создавать объект этого типа, но синтаксис A.a.offsetof не компилируется. Здесь поможет такой трюк: выражение A.init.a.offsetof позволяет получить смещение для любого внутреннего элемента структуры в виде константы, известной во время компиляции.

```
import std.stdio;

struct A {
   char a;
   int b;
   char c;
}

void main() {
   // Получить доступ к смещениям полей, не создавая объект
   writefln("%s %s %s", A.init.a.offsetof,
   A.init.b.offsetof, A.init.c.offsetof);
}
```

D гарантирует, что все байты отступов последовательно заполняются нулями.

7.1.11.1. Атрибут align

Чтобы перекрыть выбор компилятора, определив собственное выравнивание, что повлияет на вставляемые отступы, объявляйте поля с атрибутом align. Такое переопределение может понадобиться для взаимодействия с определенной аппаратурой или для работы по бинарному протоколу, задающему особое выравнивание. Пример атрибута align в действии:

```
class A {
   char a;
   align(1) int b;
```

```
    char c;
}
```

При таком определении поля структуры A располагаются без пустот между ними. (В конце объекта при этом может оставаться зарезервированное, но не занятое место.) Аргумент атрибута align означает *максимальное* выравнивание поля, но реальное выравнивание не может превысить естественное выравнивание для типа этого поля. Получить естественное выравнивание типа T позволяет определенное компилятором свойство T.alignof. Если вы, например, укажете для b выравнивание align(200) вместо указанного в примере align(1), то реально выравнивание примет значение 4, равное int.alignof.

Атрибут align можно применять к целому классу или структуре:

```
align(1) struct A {
    char a;
    int b;
    char c;
}
```

Для структуры атрибут align устанавливает выравнивание по умолчанию заданным значением. Это умолчание можно переопределить индивидуальными атрибутами align внутри структуры. Если для поля типа T указать только ключевое слово align без числа, компилятор прочитает это как align(T.alignof), то есть такая запись переустанавливает выравнивание поля в его естественное значение.

Атрибут align не предназначен для использования с указателями и ссылками. Сборщик мусора действует из расчета, что все ссылки и указатели выровнены по размеру типа size_t. Компилятор не настаивает на соблюдении этого ограничения, поскольку в общем случае у вас могут быть указатели и ссылки, не контролируемые сборщиком мусора. Таким образом, следующее определение крайне опасно, поскольку компилируется без предупреждений:

```
struct Node {
    short value;
    align(2) Node* next; // Избегайте таких определений
}
```

Если этот код выполнит присваивание объект.next = new Node (то есть заполнит объект.next ссылкой, контролируемой сборщиком мусора), хаос обеспечен: неверно выровненная ссылка пропадает из поля зрения сборщика мусора, память может быть освобождена, и объект.next превращается в «висячий» указатель.

7.2. Объединение

Объединения в стиле C можно использовать и в D, но не забывайте, что делать это нужно редко и с особой осторожностью.

Объединение (union) – это что-то вроде структуры, все внутренние поля которой начинаются по одному и тому же адресу. Таким образом, их области памяти перекрываются, а это значит, что именно вы как пользователь объединения отвечаете за соответствие записываемой и считываемой информации: нужно всегда читать в точности тот тип, который был записан. В любой конкретный момент времени только один внутренний элемент объединения обладает корректным значением.

```
union IntOrFloat {
   int _int;
   float _float;
}

unittest {
   IntOrFloat iof;
   iof._int = 5;
   // Читать только iof._int, но не iof._float
   assert(iof._int == 5);
   iof._float = 5.5;
   // Читать только iof._float, но не iof._int
   assert(iof._float == 5.5);
}
```

Поскольку типы int и float имеют строго один и тот же размер (4 байта), внутри объединения IntOrFloat их области памяти в точности совпадают. Но детали их расположения не регламентированы, например, представления _int и _float могут отличаться порядком хранения байтов: старший байт _int может иметь наименьший адрес, а старший байт _float (тот, что содержит знак и большую часть показателя степени) – наибольший адрес.

Объединения не помечаются, то есть сам объект типа union не содержит «метки», которая служила бы средством, позволяющим определять, какой из внутренних элементов является «хорошим». Ответственность за корректное использование объединения целиком ложится на плечи пользователя, что делает объединения довольно неприятным средством при построении более крупных абстракций.

В определенном, но неинициализированном объекте типа union уже есть одно инициализированное поле: первое поле автоматически инициализируется соответствующим значением .init, поэтому оно доступно для чтения сразу по завершении построения по умолчанию. Чтобы инициализировать первое поле значением, отличным от .init, укажите нужное инициализирующее выражение в фигурных скобках:

```
unittest {
   IntOrFloat iof = { 5 };
   assert(iof._int == 5);
}
```

В статическом объекте типа union может быть инициализировано и другое поле. Для этого используйте следующий синтаксис:

```
unittest {
   static IntOrFloat iof = { _float : 5 };
   assert(iof._float == 5);
}
```

Следует отметить, что нередко объединение служит именно для того, чтобы считывать тип, отличный от исходно записанного, – в соответствии с порядком управления представлением, принятым в некоторой системе. По этой причине компилятор не выявляет даже те случаи некорректного использования объединений, которые может обнаружить. Например, на 32-разрядной машине Intel следующий код компилируется и даже выполнение инструкции `assert` не порождает исключений:

```
unittest {
   IntOrFloat iof;
   iof._float = 1;
   assert(iof._int == 0x3F80_0000);
}
```

Объединение может определять функции-члены и, в общем случае, любые из тех внутренних элементов, которые может определять структура, за исключением конструкторов и деструкторов.

Чаще всего (точнее, наименее редко) объединения используются в качестве анонимных членов структур. Например:

```
import std.contracts;

struct TaggedUnion {
   enum Tag { _tvoid, _tint, _tdouble, _tstring, _tarray }
   private Tag _tag;
   private union {
      int _int;
      double _double;
      string _string;
      TaggedUnion[] _array;
   }

public:
   void opAssign(int v) {
      _int = v;
      _tag = Tag._tint;
   }
   int getInt() {
      enforce(_tag == Tag._tint);
      return _int;
   }
   ...
}

unittest {
   TaggedUnion a;
```

```
    a = 4;
    assert(a.getInt() == 4);
}
```

(Подробно тип enum описан в разделе 7.3.)

Этот пример демонстрирует чисто классический способ использования union в качестве вспомогательного средства для определения так называемого *размеченного объединения* (*discriminated union, tagged union*), также известного как алгебраический тип. Размеченное объединение инкапсулирует небезопасный объект типа union в «безопасной коробке», которая отслеживает последний присвоенный тип. Сразу после инициализации поле Tag содержит значение Tag._tvoid, по сути означающее, что объект не инициализирован. При присваивании объединению некоторого значения срабатывает оператор opAssign, устанавливающий тип объекта в соответствии с типом присваиваемого значения. Чтобы получить законченную реализацию, потребуется определить методы opAssign(double), opAssign(string) и opAssign(TaggedUnion[]) с соответствующими функциями getXxx().

Внутренний элемент типа union анонимен, то есть одновременно является и определением типа, и определением внутреннего элемента. Память под анонимное объединение выделяется как под обычный внутренний элемент структуры, и внутренние элементы этого объединения напрямую видимы внутри структуры (как показывают методы TaggedUnion). В общем случае можно определять как анонимные структуры, так и анонимные объединения, и вкладывать их как угодно.

В конце концов вы должны понять, что объединение не такое уж зло, каким может показаться. Как правило, использовать объединение вместо того, чтобы играть типами с помощью выражения cast, – хороший тон в общении между программистом и компилятором. Объединение указателя и целого числа указывает сборщику мусора, что ему следует быть осторожнее и не собирать этот указатель. Если вы сохраните указатель в целом числе и будете время от времени преобразовывать его назад к типу указателя (с помощью cast), результаты окажутся непредсказуемыми, ведь сборщик мусора может забрать память, ассоциированную с этим тайным указателем.

7.3. Перечисляемые значения

Типы, принимающие всего несколько определенных значений, оказались очень полезными – настолько полезными, что язык Java после нескольких лет героических попыток эмулировать перечисляемые типы с помощью идиомы в конце концов добавил их к основным типам [8]. Определить хорошие перечисляемые типы непросто – в C (и особенно в C++) типу enum присущи свои странности. D попытался учесть предшествующий опыт, определив простое и полезное средство для работы с перечисляемыми типами.

Начнем с азов. Простейший способ применить enum – как сказать «давайте перечислим несколько символьных значений», не ассоциируя их с новым типом:

```
enum
   mega = 1024 * 1024,
   pi = 3.14,
   euler = 2.72,
   greet = "Hello";
```

С enum механизм автоматического определения типа работает так же, как и с auto, поэтому в нашем примере переменные pi и euler имеют тип double, а переменная greet – тип string. Чтобы определить одно или несколько перечисляемых значений определенного типа, укажите их справа от ключевого слова enum:

```
enum float verySmall = 0.0001, veryBig = 10000;
enum dstring wideMsg = "Wide load";
```

Перечисляемые значения – это константы; они практически эквивалентны литералам, которые обозначают. В частности, поддерживают те же операции – например, невозможно получить адрес pi, как невозможно получить адрес 3.14:

```
auto x = pi;          // Все в порядке, x обладает типом double
auto y = pi * euler; // Все в порядке, y обладает типом double
euler = 2.73;         // Ошибка!
                      // Невозможно изменить перечисляемое значение!
void fun(ref double x) {

}
fun(pi);              // Ошибка!
                      // Невозможно получить адрес 3.14!
```

Как показано выше, типы перечисляемых значений не ограничиваются типом int – типы double и string также допустимы. Какие вообще типы можно использовать с enum? Ответ прост: с enum можно использовать любой основной тип и любую структуру. Есть лишь два требования к инициализирующему значению при определении перечисляемых значений:

- инициализирующее значение должно быть вычислимым во время компиляции;
- тип инициализирующего значения должен позволять копирование, то есть в его определении не должно быть @disable this(this) (см. раздел 7.1.3.4).

Первое требование гарантирует независимость перечисляемого значения от параметров времени исполнения. Второе требование обеспечивает возможность копировать значение; копия создается при каждом обращении к перечисляемому значению.

Невозможно определить перечисляемое значение типа class, поскольку объекты классов должны всегда создаваться с помощью оператора new

(за исключением не представляющего интерес значения null), а выражение с new во время компиляции вычислить невозможно. Не будет неожиданностью, если в будущем это ограничение снимут или ослабят.

Создадим перечисление значений типа struct:

```
struct Color {
   ubyte r, g, b;
}
enum
   red = Color(255, 0, 0),
   green = Color(0, 255, 0),
   blue = Color(0, 0, 255);
```

Когда бы вы ни использовали, например, идентификатор green, код будет вести себя так, будто вместо этого идентификатора вы написали Color(0, 255, 0).

7.3.1. Перечисляемые типы

Можно дать имя группе перечисляемых значений, создав таким образом новый тип на ее основе:

```
enum OddWord { acini, alembicated, prolegomena, aprosexia }
```

Члены именованной группы перечисляемых значений не могут иметь разные типы; все перечисляемые значения должны иметь один и тот же тип, поскольку пользователи могут впоследствии определять и использовать значения этого типа. Например:

```
OddWord w;
assert(w == OddWord.acini); // Инициализирующим значением по умолчанию
                            // является первое значение в множестве - acini.
w = OddWord.aprosexia;      // Всегда уточняйте имя значения
                            // (кстати, это не то, что вы могли подумать)
                            // с помощью имени типа.
int x = w;                  // OddWord конвертируем в int, но не наоборот.
assert(x == 3);             // Значения нумеруются по порядку: 0, 1, 2,
```

Тип, автоматически определяемый для поименованного перечисления, – int. Присвоить другой тип можно так:

```
enum OddWord : byte { acini, alembicated, prolegomena, aprosexia }
```

С новым определением (byte называют *базовым типом* OddWord) значения идентификаторов перечисления не меняются, изменяется лишь способ их хранения. Вы можете с таким же успехом назначить членам перечисления тип double или real, но связанные с идентификаторами значения останутся прежними: 0, 1 и т. д. Но если сделать базовым типом OddWord нечисловой тип, например string, то придется указать инициализирующее значение для каждого из значений, поскольку компилятору неизвестна никакая естественная последовательность, которой он мог бы придерживаться.

Возвратимся к числовым перечислениям. Присвоив какому-либо члену перечисления особое значение, вы таким образом сбросите счетчик, используемый компилятором для присваивания значений идентификаторам. Например:

```
enum E { a, b = 2, c, d = -1, e, f }
assert(E.c == 3);
assert(E.e == 0);
```

Если два идентификатора перечисления получают одно и то же значение (как в случае с E.a и E.e), конфликта нет. Фактически равные значения можно создавать, даже не подозревая об этом – из-за непреодолимого желания типов с плавающей запятой удивить небдительных пользователей:

```
enum F : float { a = 1E30, b, c, d }
assert(F.a == F.d); // Тест пройден
```

Корень этой проблемы в том, что наибольшее значение типа int, которое может быть представлено значением типа float, равно 16_777_216, и выход за эту границу сопровождается все возрастающими диапазонами целых значений, представляемых одним и тем же числом типа float.

7.3.2. Свойства перечисляемых типов

Для всякого перечисляемого типа E определены три свойства: E.init (это свойство принимает первое из значений, определенных в E), E.min (наименьшее из определенных в E значений) и E.max (наибольшее из определенных в E значений). Два последних значения определены, только если базовым типом E является тип, поддерживающий сравнение во время компиляции с помощью оператора <.

Вы вправе определить внутри enum собственные значения min, max и init, но поступать так не рекомендуется: обобщенный код частенько рассчитывает на то, что эти значения обладают особой семантикой.

Один из часто задаваемых вопросов: «Можно ли добраться до имени перечисляемого значения?» Вне всяких сомнений, сделать это возможно и на самом деле легко, но не с помощью встроенного механизма, а на основе рефлексии времени компиляции. Рефлексия работает так: с некоторым перечисляемым типом Enum связывается известная во время компиляции константа __traits(allMembers, Enum), которая содержит все члены Enum в виде кортежа значений типа string. Поскольку строками можно манипулировать во время компиляции, как и во время исполнения, такой подход дает значительную гибкость. Например, немного забежав вперед, напишем функцию toString, которая возвращает строку, соответствующую заданному перечисляемому значению. Функция параметризирована перечисляемым типом.

```
string toString(E)(E value) if (is(E == enum)) {
   foreach (s; __traits(allMembers, E)) {
```

```
        if (value == mixin("E." ~ s)) return s;
    }
    return null;
}

enum OddWord { acini, alembicated, prolegomena, aprosexia }

void main() {
    auto w = OddWord.alembicated;
    assert(toString(w) == "alembicated");
}
```

Незнакомое пока выражение `mixin("E." ~ s)` — это *выражение `mixin`*. Выражение `mixin` принимает строку, известную во время компиляции, и просто вычисляет ее как обычное выражение в рамках текущего контекста. В нашем примере это выражение включает имя перечисления `E`, оператор для выбора внутренних элементов и переменную `s` для перебора идентификаторов перечисляемых значений. В данном случае `s` последовательно принимает значения `"acini"`, `"alembicated"`, ..., `"aprosexia"`. Таким образом, конкатенированная строка примет вид `"E.acini"` и т. д., а выражение `mixin` вычислит ее, сопоставив указанным идентификаторам реальные значения. Обнаружив, что переданное значение равно очередному значению, вычисленному выражением `mixin`, функция `toString` возвращает результат. Получив некорректный аргумент `value`, функция `toString` могла бы порождать исключение, но чтобы упростить себе жизнь, мы решили просто возвращать константу `null`.

Рассмотренная функция `toString` уже реализована в модуле `std.conv` стандартной библиотеки, имеющем дело с общими преобразованиями. Имя этой функции немного отличается от того, что использовали мы: вам придется писать `to!string(w)` вместо `toString(w)`, что говорит о гибкости этой функции (также можно сделать вызов `to!dstring(w)` или `to!byte(w)` и т. д.). Этот же модуль определяет и обратную функцию, которая конвертирует строку в значение перечисляемого типа; например вызов `to!OddWord("acini")` возвращает `OddWord.acini`.

7.4. alias

В ряде случаев мы уже имели дело с `size_t` — целым типом без знака, достаточно вместительным, чтобы представить размер любого объекта. Тип `size_t` не определен языком, он просто принимает форму `uint` или `ulong` в зависимости от адресного пространства конечной системы (32 или 64 бита соответственно).

Если бы вы открыли файл `object.di`, один из копируемых на компьютер пользователя (а значит, и на ваш) при инсталляции компилятора D, то нашли бы объявление примерно следующего вида:

```
alias typeof(int.sizeof) size_t;
```

Свойство .sizeof точно измеряет размер типа в байтах; в данном случае это тип int. Вместо int в примере мог быть любой другой тип; в данном случае имеет значение не указанный тип, а тип размера, возвращаемый оператором typeof. Компилятор измеряет размеры объектов, используя uint на 32-разрядных архитектурах и ulong на 64-разрядных. Следовательно, конструкция alias позволяет назначить size_t синонимом uint или ulong.

Обобщенный синтаксис объявления с ключевым словом alias ничуть не сложнее приведенного выше:

```
alias ‹существующийИдентификатор› ‹новыйИдентификатор›;
```

В качестве идентификатора *‹существующийИдентификатор›* можно подставить все, у чего есть имя. Это может быть тип, переменная, модуль – если что-то обладает идентификатором, то для этого объекта можно создать псевдоним. Например:

```
import std.stdio;

void fun(int) {}
void fun(string) {}
int var;
enum E { e }
struct S { int x; }
S s;

unittest {
   alias object.Object Root; // Предок всех классов
   alias std phobos;         // Имя пакета
   alias std.stdio io;       // Имя модуля
   alias var sameAsVar;      // Переменная
   alias E MyEnum;           // Перечисляемый тип
   alias E.e myEnumValue;    // Значение этого типа
   alias fun gun;            // Перегруженная функция
   alias S.x field;          // Поле структуры
   alias s.x sfield;         // Поле объекта
}
```

Правила применения псевдонима просты: используйте псевдоним везде, где допустимо использовать исходный идентификатор. Именно это делает компилятор, но с точностью до наоборот: он с пониманием заменяет идентификатор-псевдоним оригинальным идентификатором. Даже сообщения об ошибках и отлаживаемая программа могут «видеть сквозь» псевдонимы и показывать исходные идентификаторы, что может оказаться неожиданным. Например, в некоторых сообщениях об ошибках или в отладочных символах можно увидеть immutable(char)[] вместо string. Но что именно будет показано, зависит от реализации компилятора.

С помощью конструкции alias можно создавать псевдонимы псевдонимов для идентификаторов, уже имеющих псевдонимы. Например:

```
alias int Int;
alias Int MyInt;
```

Здесь нет ничего особенного, просто следование обычным правилам: к моменту определения псевдонима MyInt псевдоним Int уже будет заменен исходным идентификатором int, для которого Int является псевдонимом.

Конструкцию alias часто применяют, когда требуется дать сложной цепочке идентификаторов более короткое имя или в связке с перегруженными функциями из разных модулей (см. раздел 5.5.2).

Также конструкцию alias часто используют с параметризированными структурами и классами. Например:

```
// Определить класс-контейнер
class Container(T) {
   alias T ElementType;

}

unittest {
   Container!int container;
   Container!int.ElementType element;
   ..
}
```

Здесь общедоступный псевдоним ElementType, созданный классом Container, – единственный разумный способ обратиться из внешнего мира к аргументу, привязанному к параметру T класса Container. Идентификатор T видим лишь внутри определения класса Container, но не снаружи: выражение Container!int.T не компилируется.

Наконец, конструкция alias весьма полезна в сочетании с конструкцией static if. Например:

```
// Из файла object.di
// Определить тип разности между двумя указателями
static if (size_t.sizeof == 4) {
   alias int ptrdiff_t;
} else {
   alias long ptrdiff_t;
}
// Использовать ptrdiff_t
```

С помощью объявления псевдоним ptrdiff_t привязывается к разным типам в зависимости от того, по какой ветке статического условия пойдет поток управления. Без этой возможности привязки код, которому потребовался такой тип, пришлось бы разместить в одной из веток static if.

7.5. Параметризированные контексты (конструкция template)

Мы уже рассмотрели средства, облегчающие параметризацию во время компиляции (эти средства сродни шаблонам из C++ и родовым типам из языков Java и C#), – это функции (см. раздел 5.3), параметризированные классы (см. раздел 6.14) и параметризированные структуры, которые подчиняются тем же правилам, что и параметризированные классы. Тем не менее иногда во время компиляции требуется каким-либо образом манипулировать типами, не определяя функцию, структуру или класс. Один из механизмов, подходящих под это описание (широко используемый в C++), – выбор того или иного типа в зависимости от статически известного логического условия. При этом не определяется никакой новый тип и не вызывается никакая функция – лишь создается псевдоним для одного из существующих типов.

Для случаев, когда требуется организовать параметризацию во время компиляции без определения нового типа или функции, D предоставляет параметризированные контексты. Такой параметризированный контекст вводится следующим образом:

```
template Select(bool cond, T1, T2) {
   ...
}
```

Этот код – на самом деле лишь каркас для только что упомянутого механизма выбора во время компиляции. Скоро мы доберемся и до реализации, а пока сосредоточимся на порядке объявления. Объявление с ключевым словом `template` вводит именованный контекст (в данном случае это `Select`) с параметрами, вычисляемыми во время компиляции (в данном случае это логическое значение и два типа). Объявить контекст можно на уровне модуля, внутри определения класса, внутри определения структуры, внутри любого другого объявления контекста, но не внутри определения функции.

В теле параметризированного контекста разрешается использовать все те же объявления, что и обычно, кроме того, могут быть использованы параметры контекста. Доступ к любому объявлению контекста можно получить извне, расположив перед его именем имя контекста и ., например: `Select!(true, int, double).foo`. Давайте прямо сейчас закончим определение контекста `Select`, чтобы можно было поиграть с ним:

```
template Select(bool cond, T1, T2) {
   static if (cond) {
      alias T1 Type;
   } else {
      alias T2 Type;
   }
}
```

```
unittest {
   alias Select!(false, int, string).Type MyType;
   static assert(is(MyType == string));
}
```

Заметим, что тот же результат мы могли бы получить на основе структуры или класса, поскольку эти типы могут определять в качестве своих внутренних элементов псевдонимы, доступные с помощью обычного синтаксиса с оператором . (точка):

```
struct /* или class */ Select2(bool cond, T1, T2) {
   static if (cond) {
      alias T1 Type;
   } else {
      alias T2 Type;
   }
}

unittest {
   alias Select2!(false, int, string).Type MyType;
   static assert(is(MyType == string));
}
```

Согласитесь, такое решение выглядит не очень привлекательно. К примеру, для Select2 в документации пришлось бы написать: «Не создавайте объекты типа Select2! Он определен только ради псевдонима внутри него!» Доступный специализированный механизм определения параметризированных контекстов позволяет избежать двусмысленности намерений, не вызывает недоумения и исключает возможность некорректного использования.

В контексте, определенном с ключевым словом template, можно объявлять не только псевдонимы – там могут присутствовать самые разные объявления. Определим еще один полезный шаблон. На этот раз это будет шаблон, возвращающий логическое значение, которое сообщает, является ли заданный тип строкой (в любой кодировке):

```
template isSomeString(T) {
   enum bool value = is(T : const(char[]))
      || is(T : const(wchar[])) || is(T : const(dchar[]));
}

unittest {
   // Не строки
   static assert(!isSomeString!(int).value);
   static assert(!isSomeString!(byte[]).value);
   // Строки
   static assert(isSomeString!(char[]).value);
   static assert(isSomeString!(dchar[]).value);
   static assert(isSomeString!(string).value);
   static assert(isSomeString!(wstring).value);
   static assert(isSomeString!(dstring).value);
```

```
    static assert(isSomeString!(char[4]).value);
}
```

Параметризированные контексты могут быть рекурсивными; к примеру, вот одно из возможных решений задачи с факториалом:

```
template factorial(uint n) {
    static if (n <= 1)
        enum ulong value = 1;
    else
        enum ulong value = factorial!(n - 1).value * n;
}
```

Несмотря на то что factorial является совершенным функциональным определением, в данном случае это не лучший подход. При необходимости вычислять значения во время компиляции, пожалуй, стоило бы воспользоваться механизмом вычислений во время компиляции (см. раздел 5.12). В отличие от приведенного выше шаблона factorial, функция factorial более гибка, поскольку может вычисляться как во время компиляции, так и во время исполнения. Конструкция template больше всего подходит для манипуляции типами, имеющей место в Select и isSomeString.

7.5.1. Одноименные шаблоны

Конструкция template может определять любое количество идентификаторов, но, как видно из предыдущих примеров, нередко в ней определен ровно один идентификатор. Обычно шаблон определяется лишь с целью решить единственную задачу и в качестве результата сделать доступным единственный идентификатор (такой как Type в случае Select или value в случае isSomeString).

Необходимость помнить о том, что в конце вызова надо указать этот идентификатор, и всегда его указывать может раздражать. Многие просто забывают добавить в конец .Type, а потом удивляются, почему вызов Select!(cond, A, B) порождает таинственное сообщение об ошибке.

D помогает здесь, определяя правило, известное как фокус с одноименным шаблоном: если внутри конструкции template определен идентификатор, совпадающий с именем самого шаблона, то при любом последующем использовании имени этого шаблона в его конец будет автоматически дописываться одноименный идентификатор. Например:

```
template isNumeric(T) {
    enum bool isNumeric = is(T : long) || is(T : real);
}

unittest {
    static assert(isNumeric!(int));
    static assert(!isNumeric!(char[]));
}
```

Если теперь некоторый код использует выражение isNumeric!(T), компилятор в каждом случае автоматически заменит его на isNumeric!(T).isNumeric, чем освободит пользователя от хлопот с добавлением идентификатора в конец имени шаблона.

Шаблон, проделывающий фокус с «тезками», может определять внутри себя и другие идентификаторы, но они будут попросту недоступны за пределами этого шаблона. Дело в том, что компилятор заменяет идентификаторы на раннем этапе процесса поиска имен. Единственный способ получить доступ к таким идентификаторам – обратиться к ним из тела самого шаблона. Например:

```
template isNumeric(T) {
   enum bool test1 = is(T : long);
   enum bool test2 = is(T : real);
   enum bool isNumeric = test1 || test2;
}

unittest {
   static assert(isNumeric!(int).test1); // Ошибка!
   // Тип bool не определяет свойство test1!
}
```

Это сообщение об ошибке вызвано соблюдением правила об одноименности: перед тем как делать что-либо еще, компилятор расширяет вызов isNumeric!(int) до isNumeric!(int).isNumeric. Затем пользовательский код делает попытку заполучить значение isNumeric!(int).isNumeric.test1, что равносильно попытке получить внутренний элемент test1 из логического значения, отсюда и сообщение об ошибке. Короче говоря, используйте одноименные шаблоны тогда и только тогда, когда хотите открыть доступ лишь к одному идентификатору. Этот случай скорее частый, чем редкий, поэтому одноименные шаблоны очень популярны и удобны.

7.5.2. Параметр шаблона this[1]

Познакомившись с классами и структурами, можно параметризовать наш обобщенный метод типом неявного аргумента this. Например:

```
class Parent
{
   static string getName(this T)()
   {
      return T.stringof;
   }

}
```

[1] На момент написания оригинала книги данная возможность отсутствовала, но поскольку теперь она существует, мы добавили ее описание в перевод. – *Прим. науч. ред.*

```
class Derived1: Parent{}
class Derived2: Parent{}

unittest
{
   assert(Parent.getName() == "Parent");
   assert(Derived1.getName() == "Derived1");
   assert(Derived2.getName() == "Derived2");
}
```

Параметр шаблона `this T` предписывает компилятору в теле `getName` считать `T` псевдонимом `typeof(this)`.

В обычный статический метод класса не передаются никакие скрытые параметры, поэтому невозможно определить, для какого конкретно класса вызван этот метод. В приведенном примере компилятор создает три экземпляра шаблонного метода `Parent.getName(this T)()`: `Parent.getName()`, `Derived1.getName()` и `Derived2.getName()`.

Также параметр `this` удобен в случае, когда один метод нужно использовать для разных квалификаторов неизменяемости объекта (см. главу 8).

7.6. Инъекции кода с помощью конструкции mixin template

При некоторых программных решениях приходится добавлять шаблонный код (такой как определения данных и методов) в одну или несколько реализаций классов. К типичным примерам относятся поддержка сериализации, шаблон проектирования «Наблюдатель» [27] и передача событий в оконных системах.

Для этих целей можно было бы воспользоваться механизмом наследования, но поскольку реализуется лишь одиночное наследование, определить для заданного класса несколько источников шаблонного кода невозможно. Иногда необходим механизм, позволяющий просто вставить в класс некоторый готовый код, вместо того чтобы писать его вручную.

Здесь-то и пригодится конструкция `mixin template` (шаблон `mixin`). Стоит отметить, что сейчас это средство в основном экспериментальное. Возможно, в будущих версиях языка шаблоны `mixin` заменит более общий инструмент AST-макросов.

Шаблон `mixin` определяется почти так же, как параметризированный контекст (шаблон), о котором недавно шла речь. Пример шаблона `mixin`, определяющего переменную и функции для ее чтения и записи:

```
mixin template InjectX() {
   private int x;
   int getX() { return x; }
   void setX(int y) {
```

```
        .. // Проверки
        x = y;
    }
}
```

Определив шаблон `mixin`, можно вставить его в нескольких местах:

```
// Сделать инъекцию в контексте модуля
mixin InjectX;

class A {
    // Сделать инъекцию в класс
    mixin InjectX;
     ..
}

void fun() {
    // Сделать инъекцию в функцию
    mixin InjectX;
    setX(10);
    assert(getX() == 10);
}
```

Теперь этот код определяет переменную и две обслуживающие ее функции на уровне модуля, внутри класса `A` и внутри функции `fun` – как будто тело `InjectX` было вставлено вручную. В частности, потомки класса `A` могут переопределять методы `getX` и `setX`, как если бы сам класс определял их. Копирование и вставка без неприятного дублирования кода – вот что такое `mixin template`.

Конечно же, следующий логический шаг – подумать о том, что `InjectX` не принимает никаких параметров, но производит впечатление, что мог бы, – и действительно может:

```
mixin template InjectX(T) {
    private T x;
    T getX() { return x; }
    void setX(T y) {
        .. // Проверки
        x = y;
    }
}
```

Теперь при обращении к `InjectX` нужно передавать аргумент так:

```
mixin InjectX!int;
mixin InjectX!double;
```

Но на самом деле такие вставки приводят к двусмысленности: что если вы сделаете две рассмотренные подстановки, а затем пожелаете воспользоваться функцией `getX`? Есть две функции с этим именем, так что проблема с двусмысленностью очевидна. Чтобы решить этот вопрос, D позволяет вводить *имена* для конкретных подстановок в шаблоны `mixin`:

```
mixin InjectX!int MyInt;
mixin InjectX!double MyDouble;
```

Задав такие определения, вы можете недвусмысленно обратиться к внутренним элементам любого из шаблонов mixin, просто указав нужный контекст:

```
MyInt.setX(5);
assert(MyInt.getX() == 5);
MyDouble.setX(5.5);
assert(MyDouble.getX() == 5.5);
```

Таким образом, шаблоны mixin – это *почти* как копирование и вставка; вы можете многократно копировать и вставлять код, а потом указывать, к какой именно вставке хотите обратиться.

7.6.1. Поиск идентификаторов внутри mixin

Самая большая разница между шаблоном mixin и обычным шаблоном (в том виде, как он определен в разделе 7.5), способная вызвать больше всего вопросов, – это поиск имен.

Шаблоны исключительно модульны: код внутри шаблона ищет идентификаторы в месте *определения* шаблона. Это положительное качество: оно гарантирует, что, проанализировав определение шаблона, вы уже ясно представляете его содержимое и осознаете, как он работает.

Шаблон mixin, напротив, ищет идентификаторы в месте *подстановки*, а это означает, что понять поведение шаблона mixin можно только с учетом контекста, в котором вы собираетесь этот шаблон использовать.

Чтобы проиллюстрировать разницу, рассмотрим следующий пример, в котором идентификаторы объявляются как в месте определения, так и в месте подстановки:

```
import std.stdio;

string lookMeUp = "Найдено на уровне модуля";

template TestT() {
   string get() { return lookMeUp; }
}

mixin template TestM() {
   string get() { return lookMeUp; }
}

void main() {
   string lookMeUp = "Найдено на уровне функции";
   alias TestT!() asTemplate;
   mixin TestM!() asMixin;
   writeln(asTemplate.get());
   writeln(asMixin.get());
}
```

Эта программа выведет на экран:

```
Найдено на уровне модуля
Найдено на уровне функции
```

Склонность шаблонов `mixin` привязываться к локальным идентификаторам придает им выразительности, но следовать их логике становится сложно. Такое поведение делает шаблоны `mixin` применимыми лишь в ограниченном количестве случаев; прежде чем доставать из ящика с инструментами эти особенные ножницы, необходимо семь раз отмерить.

7.7. Итоги

Классы позволяют эффективно представить далеко не любую абстракцию. Например, они не подходят для мелкокалиберных объектов, контекстно-зависимых ресурсов и типов значений. Этот пробел восполняют структуры. В частности, благодаря конструкторам и деструкторам легко определять типы контекстно-зависимых ресурсов.

Объединения – низкоуровневое средство, позволяющее хранить разные типы данных в одной области памяти с перекрыванием.

Перечисления – это обычные отдельные значения, определенные пользователем. Перечислению может быть назначен новый тип, что позволяет более точно проверять типы значений, определенных в рамках этого типа.

`alias` – очень полезное средство, позволяющее привязать один идентификатор к другому. Нередко псевдоним – единственное средство получить извне доступ к идентификатору, вычисляемому в рамках вложенной сущности, или к длинному и сложному идентификатору.

Параметризированные контексты, использующие конструкцию `template`, весьма полезны для определения вычислений во время компиляции, таких как интроспекция типов и определение особенностей типов. Одноименные шаблоны позволяют предоставлять абстракции в очень удобной, инкапсулированной форме.

Кроме того, предлагаются параметризированные контексты, принимающие форму шаблонов `mixin`, которые во многом ведут себя подобно макросам. В будущем шаблоны `mixin` может заменить развитое средство AST-макросов.

8

Квалификаторы типа

Квалификаторы типа выражают важные утверждения о типах языка. Эти утверждения исключительно полезны как для программиста, так и для компилятора, но их сложно выразить путем соглашений, обычного порождения подтипов (см. раздел 6.4.2) или параметризации типами (см. раздел 6.14).

Показательный пример квалификатора типа – квалификатор типа `const` (введенный в языке C и доработанный в C++). Примененный к типу `T`, этот квалификатор выражает следующее утверждение: значения типа `T` можно инициализировать и читать, но не перезаписывать. Соблюдение этого ограничения гарантируется компилятором. Квалификатор `const` довольно полезен внутри модуля, поскольку гарантирует инициаторам вызовов регламентированное поведение функций. Например, сигнатура

```
// Функция из стандартной библиотеки C
int printf(const char * format, . );
```

обещает пользователям, что функция `printf` не будет пытаться изменить знаки, переданные в параметре `format`. Подобная гарантия также полезна при масштабной разработке, поскольку сокращает количество зависимостей, созданных немодульными изменениями. Определить такие ограничения и гарантировать подчинение им можно и посредством соглашения, но подобные соглашения неудобны, и соблюдать их трудно.

D определяет три типа квалификаторов:

- `const` означает неизменяемость в рамках заданного контекста. Значение типа, заданного с ключевым словом `const`, нельзя изменить напрямую. Однако другие сущности в программе могут обладать правом перезаписывать эти данные: так у инициатора вызова функции `printf` может быть право записи в переменную `format`, а у самой функции – нет.

- immutable означает абсолютную, контекстно-независимую неизменяемость. Значение типа, заданного с ключевым словом immutable, после инициализации нельзя изменить ни при каких обстоятельствах нигде в программе. Это гораздо более строгое ограничение, чем у квалификатора const.
- shared означает разделение значения между потоками.

Все они дополняют друг друга. Квалификаторы const и immutable важны для масштабной разработки. Кроме того, без квалификатора immutable невозможно было бы программировать в функциональном стиле, а квалификатор const способствует интеграции кода в функциональном стиле с кодом в объектно-ориентированном и процедурном стиле. Квалификаторы immutable и shared позволяют реализовать многопоточность. Подробное описание квалификатора shared и разговор о многопоточности мы отложим до главы 13. А здесь сосредоточимся на квалификаторах const и immutable.

8.1. Квалификатор immutable

Значение типа с квалификатором immutable высечено на камне: сразу же после инициализации такого значения можно считать, что оно навечно прожжено в хранящей его памяти. Оно никогда не изменится за все время исполнения программы.

Форма записи типа с квалификатором такова: *‹квалификатор›*(T), где *‹квалификатор›* — одно из ключевых слов immutable, const и shared. Например, определим неизменяемое целое число:

```
immutable(int) forever = 42;
```

Попытки каким-либо способом изменить значение переменной forever приведут к ошибке во время компиляции. Более того, immutable(int) — это полноправный тип, как любой другой тип (он отличается от типа int). Например, можно присвоить ему псевдоним:

```
alias immutable(int) StableInt;
StableInt forever = 42;
```

Определяя копию переменной forever с ключевым словом auto, вы распространите тип immutable(int) и на копию, так что и сама копия будет неизменяемым целым числом. Ничего особенного здесь нет, но именно этим отличаются квалификаторы типов и простые классы памяти, такие как static (см. раздел 5.2.5) или ref (см. раздел 5.2.1).

```
unittest {
   immutable(int) forever = 42;
   auto andEver = forever;
   ++andEver; // Ошибка! Нельзя изменять неизменяемое значение!
}
```

Значение типа с квалификатором immutable необязательно инициализировать константой, известной во время компиляции:

```
void fun(int x) {
   immutable(int) xEntry = x;

}
```

Примененный таким образом квалификатор immutable оказывает услугу тем, кто будет разбираться в работе функции fun. С первого взгляда понятно, что переменная xEntry будет хранить переданное на входе в функцию значение x от начала и до конца тела этой функции.

В определениях с квалификатором immutable необязательно указывать тип – он будет определен так же, как если бы вместо immutable стояло ключевое слово auto:

```
immutable pi = 3.14, val = 42;
```

Для pi компилятор выводит тип immutable(double), а для val – immutable(int).

8.1.1. Транзитивность

Любой тип можно определить с квалификатором immutable. Например:

```
struct Point { int x, y; }
auto origin = immutable(Point)(0, 0);
```

Поскольку для всех типов T справедливо, что immutable(T) – такой же тип, как любой другой, запись immutable(Point)(0, 0) является литералом структуры – так же как и Point(0, 0).

Неизменяемость естественным образом распространяется на все внутренние элементы объекта. Ведь пользователь ожидает, что если запрещено присваивание объекту origin в целом, то запрещено и присваивание полям origin.x и origin.y. Иначе было бы очень легко нарушить ограничение, налагаемое квалификатором immutable на объект в целом.

```
unittest {
   auto anotherOrigin = immutable(Point)(1, 1);
   origin = anotherOrigin; // Ошибка!
   origin.x = 1;           // Ошибка!
   origin.y = 1;           // Ошибка!
}
```

На самом деле, immutable *распространяется* абсолютно на каждое поле Point, тип каждого поля объекта квалифицируется тем же квалификатором, что и сам объект. Например, такой тест будет пройден:

```
static assert(is(typeof(origin.x) == immutable(int))); // Тест пройден
```

Но мир не настолько прост. Рассмотрим структуру, в которой есть некоторая косвенность, например поле массива:

```
struct DataSample {
   int id;
   double[] payload;
}
```

Очевидно, что поля объекта типа `immutable(DataSample)` не могут быть изменены. Но как насчет изменения элемента массива `payload`?

```
unittest {
   auto ds = immutable(DataSample)(5, [ 1.0, 2.0 ]);
    ds.payload[1] = 4.5; // ?
}
```

В данном случае может быть принято одно из двух возможных решений, у каждого из которых есть свои плюсы и минусы. Один из вариантов – сделать действие квалификатора *поверхностным*, то есть руководствоваться тем, что квалификатор `immutable`, примененный к структуре `DataSample`, применяется и ко всем ее непосредственным полям, но никак не влияет на данные, косвенно доступные через эти поля[1]. Альтернативное решение – сделать неизменяемость *транзитивной*, что означало бы следующее: делая объект неизменяемым, вы также делаете неизменяемыми все данные, к которым можно обратиться через этот объект. Язык D пошел по второму пути.

Транзитивная неизменяемость гораздо строже нетранзитивной. Определив неизменяемое значение, вы накладываете ограничение неизменяемости на целую сеть данных, связанную с этим значением (через ссылки, массивы и указатели). Таким образом, определять транзитивно неизменяемые значения гораздо сложнее, чем поверхностно неизменяемые. Но приложенные усилия окупаются сторицей. D выбрал транзитивную изменяемость по двум основным причинам:

- *Функциональное программирование.* Словосочетание «функциональный стиль» каждый интерпретирует по-своему, но большинство согласятся, что отсутствие побочных эффектов – важный принцип. Обеспечить соблюдение этого ограничения лишь соглашением – значит отказаться от масштабируемости. Транзитивная неизменяемость позволяет программисту применять функциональный стиль для хорошо определенного фрагмента программы, а компилятору – проверять этот функциональный код на предмет непреднамеренных изменений данных.
- *Параллельное программирование.* Параллелизм – это огромная и очень сложная тема, занимаясь которой, ощущаешь нехватку твердых гарантий и неоспоримых истин. Неизменяемое разделение – один из таких островков уверенности: разделение неизменяемых данных между потоками всегда корректно, безопасно и эффективно. Чтобы позволить компилятору проверять, не изменяемы ли на самом деле разделяемые данные, неизменяемость должна быть транзитивной;

[1] Такой подход был избран для квалификатора `const` в C++.

в противном случае поток, обладающий доступом к неизменяемому фрагменту данных, может легко перейти к изменяемому разделению, попросту обратившись к косвенным полям этого фрагмента данных.

Получив значение типа `immutable(T)`, можно быть абсолютно уверенным, что все, до чего можно добраться через это значение, также квалифицировано с помощью `immutable`, то есть неизменяемо. Более того, *никто* никогда не сможет изменить эти данные – данные, помеченные квалификатором `immutable`, все равно что впаяны. Это очень надежная гарантия, позволяющая, к примеру, беззаботно разделять такие неизменяемые данные между потоками.

8.2. Составление типов с помощью immutable

Учитывая, что для точного выбора типа, который нужно квалифицировать, с квалификаторами используют скобки и что тип с квалификатором – это полноправный новый тип, можно сделать вывод, что, комбинируя ключевое слово `immutable` с другими конструкторами типов, можно создавать весьма сложные структуры данных. Сравним, например, следующие два типа:

```
alias immutable(int[]) T1;
alias immutable(int)[] T2;
```

В первом определении круглые скобки поглотили полностью весь тип массива; во втором случае затронут лишь тип элементов массива `int`, но не сам массив. Если при употреблении квалификатора `immutable` круглые скобки отсутствуют, квалификатор применяется ко всему типу, так что эквивалентное определение `T1` выглядит так:

```
alias immutable int[] T1;
```

Тип `T1` незамысловат: он представляет собой неизменяемый массив значений типа `int`. Само написание этого типа говорит то же самое. В соответствии со свойством транзитивности нельзя изменить ни массив в целом (например, присвоив переменной, содержащей массив, новый массив), ни какой-либо его элемент в отдельности:

```
T1 a = [ 1, 3, 5 ];
T1 b = [ 2, 4 ];
a = b;       // Ошибка!
a[0] = b[1]; // Ошибка!
```

Второе определение кажется более тонким, но на самом деле понять его довольно просто, если вспомнить, что `immutable(int)` – самостоятельный тип. Тогда `immutable(int)[]` – это просто массив элементов этого самостоятельного типа, вот и все. Вывод о свойствах этого массива напрашивается сам. Можно присвоить значение массиву в целом, но нельзя изменить (в том числе через присваивание) отдельные его элементы:

```
T2 a = [ 1, 3, 5 ];
T2 b = [ 2, 4 ];
a = b;       // Все в порядке
a[0] = b[1]; // Ошибка!
a ~= b;      // Все в порядке (но как тонко!)
```

Может показаться странным, но добавление элементов в конец массива законно. Почему? Да просто потому, что эта операция не изменяет те элементы, которые в массиве уже есть. (Она может повлечь копирование данных, если потребуется переносить массив в другую область памяти, но в этом нет ничего страшного.)

Как уже говорилось (см. раздел 4.5), `string` — это в действительности лишь псевдоним для типа `immutable(char)[]`. На самом деле, многие из полезных свойств типа `string`, задействованных в предыдущих главах, — заслуга квалификатора `immutable`.

Сочетания ключевого слова `immutable` с параметрами-типами интерпретируются по той же логике. Предположим, есть обобщенный тип `Container!T`. Тогда в сочетании `immutable(Container!T)` квалификатор будет относиться ко всему контейнеру, а в сочетании `Container!(immutable(T))` — лишь к отдельным его элементам.

8.3. Неизменяемые параметры и методы

В сигнатуре функции квалификатор `immutable` очень информативен. Рассмотрим одну из простейших функций:

```
string process(string input);
```

На самом деле это лишь краткая запись сигнатуры

```
immutable(char)[] process(immutable(char)[] input);
```

Функция `process` гарантирует, что не будет изменять отдельные знаки параметра `input`, так что инициатор вызова функции `process` может быть уверен, что после вызова строка окажется такой же, как и перед ним:

```
string s1 = "чепуха";
string s2 = process(s1);
assert(s1 == "чепуха"); // Выполняется всегда
```

Более того, пользователь функции `process` может рассчитывать на то, что ее результат не сможет быть изменен: нет никаких скрытых псевдонимов, никакая другая функция не сможет позже изменить `s2`. В этом случае `immutable` также означает неизменяемость.

Структуры и классы могут определять неизменяемые методы. В подобных случаях квалификатор применяется к `this`:

```
class A {
   int[] fun();           // Обычный метод
   int[] gun() immutable; // Можно вызвать, только если объект неизменяемый
```

```
    immutable int[] hun(); // То же, что выше
}
```

Третья форма записи выглядит подозрительно: может показаться, что квалификатор `immutable` относится к `int[]`, но на самом деле он относится к `this`. Если нужно определить неизменяемый метод, возвращающий `immutable int[]`, получается что-то вроде заикания:

```
immutable immutable(int[]) iun();
```

Поэтому в таких случаях ключевое слово `immutable` лучше писать в конце:

```
immutable(int[]) iun() immutable;
```

Разрешение оставить несколько сбивающий с толку `immutable` в начале определения продиктовано в основном стремлением унифицировать формат указания для всех свойств методов (таких как `final` или `static`). Например, определить сразу несколько неизменяемых методов, можно так:

```
class A {
    immutable {
        int foo();
        int[] bar();
        void baz();
    }
}
```

Кроме того, квалификатор `immutable` можно использовать в виде метки:

```
class A {
    immutable:
        int foo();
        int[] bar();
        void baz();
}
```

Разумеется, неизменяемые методы могут быть вызваны только применительно к неизменяемым объектам:

```
class C {
    void fun() {}
    void gun() immutable {}
}

unittest {
    auto c1 = new C;
    auto c2 = new immutable(C);
    c1.fun(); // Все в порядке
    c2.gun(); // Все в порядке
              // Никакие другие вызовы не сработают
}
```

8.4. Неизменяемые конструкторы

Работать с неизменяемым объектом совсем несложно, а вот его построение – весьма деликатный процесс. Причина в том, что в процессе построения нужно удовлетворить два противоречивых требования: 1) присвоить полям значения, 2) сделать их неизменяемыми. Поэтому D особенно внимателен к неизменяемым конструкторам.

Проверка типов в неизменяемом конструкторе выполняется простым и осторожным способом. Компилятор разрешает присваивание полей только внутри конструктора, а чтение полей (включая передачу this в качестве аргумента при вызове метода) запрещено. Как только выполнение неизменяемого конструктора завершается, объект «замораживается» – после этого нельзя потребовать ни одного изменения. Вызов нестатического метода считается за чтение, поскольку такой метод обладает доступом к объекту this и способен прочесть любое его поле. (Компилятор не проверяет, читает ли метод поля на самом деле, – для перестраховки он предполагает, что метод все же читает некоторое поле.)

Это правило строже, чем необходимо; ведь в действительности запрещается лишь присваивание значения полю после того, как это поле было прочитано. Однако это более строгое правило практически не мешает выразительности, при этом оно простое и понятное. Например:

```
class A {
   int a;
   int[] b;
   this() immutable {
      a = 5;
      b = [ 1, 2, 3 ];
      // Вызов fun() не был бы разрешен
   }
   void fun() immutable {
      ..
   }
}
```

Вызывать из неизменяемого конструктора конструктор родителя super в порядке вещей, если этот вызов адресован также неизменяемому конструктору. Такие вызовы не угрожают нарушить неизменяемость.

Инициализировать неизменяемые объекты обычно помогает рекурсия. К примеру, рассмотрим реализующий абстракцию односвязного списка класс, инициализируемый с помощью массива:

```
class List {
   private int payload;
   private List next;
   this(int[] data) immutable {
      enforce(data.length);
      payload = data[0];
```

```
        if (data.length == 1) return;
        next = new immutable(List)(data[1    $]);
    }
}
```

Чтобы корректно инициализировать хвост списка, конструктор рекурсивно вызывает сам себя с более коротким массивом. Попытки инициализировать список в цикле не пройдут компиляцию, поскольку проход по создаваемому списку означает чтение полей, а это запрещено. Рекурсия изящно решает эту проблему[1].

8.5. Преобразования с участием immutable

Рассмотрим пример кода:

```
unittest {
    int a = 42;
    immutable(int) b = a;
    int c = b;
}
```

Более строгая система типизации не приняла бы этот код. Он включает два преобразования: сначала из `int` в `immutable(int)`, а затем обратно из `immutable(int)` в `int`. Собственно, по общим правилам эти преобразования незаконны. Например, если в этом коде заменить `int` на `int[]`, ни одно из следующих преобразований не будет корректным:

```
int[] a = [ 42 ];
immutable(int[]) b = a; // Нет!
int[] c = b;            // Нет!
```

Если бы такие преобразования были разрешены, неизменяемость бы не соблюдалась, поскольку тогда неизменяемые массивы разделяли бы свое содержимое с изменяемыми.

Тем не менее компилятор распознает и разрешает некоторые автоматические преобразования между неизменяемыми и изменяемыми данными. А именно разрешено двунаправленное преобразование между `T` и `immutable(T)`, если у `T` «нет изменяемой косвенности». Интуитивно понятно, что «нет изменяемой косвенности» означает запрет перезаписывать косвенно доступные через `T` данные. Определение этого понятия рекурсивно:

- у встроенных типов значений, таких как `int`, «нет изменяемой косвенности»;
- у массивов фиксированной длины из элементов типов, у которых «нет изменяемой косвенности», в свою очередь, тоже «нет изменяемой косвенности»;

[1] Это решение было предложено Саймоном Пейтоном-Джонсом.

- у массивов и указателей, ссылающихся на типы, у которых «нет изменяемой косвенности», тоже «нет изменяемой косвенности»;
- у структур, ни в одном поле которых «нет изменяемой косвенности», тоже «нет изменяемой косвенности».

Например, у типа S1 нет изменяемой косвенности, а у типа S2 – есть:

```
struct S1 {
   int a;
   double[3] b;
   string c;
}

struct S2 {
   int x;
   float[] y;
}
```

Из-за поля S2.y у структуры S2 есть изменяемая косвенность, так что преобразования вида immutable(S2) ↔ S2 запрещены. Если бы они были разрешены, изменяемые и неизменяемые объекты стали бы некорректно разделять данные, хранимые в y, что нарушило бы гарантии, предоставленные квалификатором immutable.

Вернемся к примеру, приведенному в начале этого раздела. Тип int не обладает изменяемой косвенностью, так что компилятор волен разрешить преобразования из int в immutable(int) и обратно.

Чтобы определить такие преобразования для структуры, вам потребуется немного поработать вручную, направляя процесс в нужное русло. Вы предоставляете соответствующие конструкторы, а компилятор обеспечивает корректность вашего кода. Проще всего одолеть преобразование, заручившись поддержкой универсальной служебной функции преобразования std.conv.to[1], которая понимает все тонкости преобразований типов с квалификаторами и всегда принимает соответственные меры.

```
import std.conv;

struct S {
   private int[] a;
   // Преобразование из неизменяемого в изменяемое
   this(immutable(S) source) {
      // Поместить дубликат массива в массив не-immutable
      a = to!(int[])(source.a);
   }
   // Преобразование из изменяемого в неизменяемое
   this(S source) immutable {
```

[1] Кроме того, у любого массива T[], const(T)[] и immutable(T)[] есть свойство dup, возвращающее копию массива типа T[], и свойство idup, возвращающее копию типа immutable(T)[]. – *Прим. науч. ред.*

```
        // Поместить дубликат массива в массив immutable
        a = to!(immutable(int[]))(source.a);
    }

}

unittest {
    S a;
    auto b = immutable(S)(a);
    auto c = S(b);
}
```

Преобразование не является неявным, но оно допустимо и безопасно.

8.6. Квалификатор const

Немного поэкспериментировав с квалификатором типа `immutable`, мы видим, что этот квалификатор слишком строг, чтобы быть полезным в большинстве случаев. Да, если вы обязались не изменять определенные данные на протяжении работы целой программы, `immutable` вам подходит. Но часто неизменяемость – свойство, полезное в рамках модуля: хотелось бы оставить за собой право изменять некоторые данные, а остальным запретить это делать. Такие данные нельзя назвать неизменяемыми (то есть пометить квалификатором `immutable`), поскольку `immutable` означает: «Видите письмена, высеченные на камне? Это ваши данные». А вам нужно средство, чтобы выразить такое ограничение: «Вы не можете изменить эти данные, но кто-кто другой – может». Или, как сказал Алан Перлис: «Кому константа, а кому и переменная». Посмотрим, как система типов вполне серьезно реализует изречение Перлиса.

Простой вариант использования таких данных – функция, например `print`, которая печатает какие-то данные. Функция `print` не изменяет переданные в нее данные, так что она допускает применение квалификатора `immutable`:

```
void print(immutable(int[]) data) {     }
unittest {
    immutable(int[]) myData = [ 10, 20 ];
    print(myData); // Все в порядке
}
```

Отлично. Далее, пусть у нас есть значение типа `int[]`, которое мы только что вычислили и хотим напечатать. С таким аргументом наша функция не сработает, поскольку значение типа `int[]` не приводится к типу `immutable(int)[]` – а если бы приводилось, то возникла бы неподобающая общность изменяемых и якобы неизменяемых данных. Получается, что функция `print` не может напечатать данные типа `int[]`. Такое ограничение довольно неоправданно, поскольку `print` вообще не затрагивает свои аргументы, так что эта функция должна работать с неизменяемыми данными так же, как с изменяемыми.

Что нам нужно, так это нечто вроде общего «либо изменяемого, либо нет» типа. В этом случае функцию print можно было бы объявить так:

```
void print(либо_изменяемый_либо_нет(int[]) data) {     }
```

Только потому, что название либо_изменяемый_либо_нет несколько длинновато, для обозначения этого типа было введено ключевое слово const. Смысл абсолютно тот же: текущий код не может изменить значение типа const(T), но есть вероятность, что это может сделать другой код. Такая двусмысленность отражает тот факт, что в роли const(T) может выступать как T, так и immutable(T). Это качество квалификатора const делает его совершенным для организации взаимодействия между функциональным и обычным процедурным кодом. Продолжим начатый выше пример[1]:

```
void print(const(int[]) data) {     }
unittest {
   immutable(int[]) myData = [ 10, 20 ];
   print(myData); // Все в порядке
   int[] myMutableData = [ 32, 42 ];
   print(myMutableData); // Все в порядке
}
```

Этот пример подразумевает, что как изменяемые, так и неизменяемые данные неявно преобразуются к const, что, в свою очередь, подразумевает нечто вроде взаимоотношения с подтипами. На самом деле, все так и есть: const(T) – это супертип и для типа T, и для типа immutable(T) (рис. 8.1).

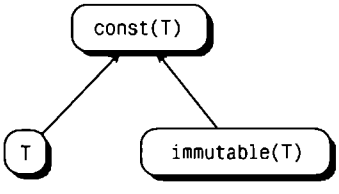

***Рис. 8.1.** Для всех типов T: const(T) – супертип и для T, и для immutable(T). Следовательно, код, работающий со значениями типа const(T), принимает значения как изменяемого, так и неизменяемого типа T*

Для квалификатора const верны те же правила транзитивности и преобразований, что и для квалификатора immutable. На конструкторы объектов const, в отличие от конструкторов immutable, ограничения не накладываются: внутри конструктора const объект считается изменяемым.

---

[1] Приведенную ниже функцию можно было бы объявить как void print(in int[] data);, что означает в точности то же самое, но несколько лучше смотрится. – *Прим. науч. ред.*

Метод, объявленный с квалификатором const, может вызываться для объектов с любым квалификатором, так как он гарантирует, что ничего менять не будет, но не требует этого от объекта. Например:

```
class C {
   void gun() const {}
}

unittest {
   auto c1 = new C;
   auto c2 = new immutable(C);
   auto c3 = new const(C);
   c1.gun(); // Все в порядке
   c2.gun(); // Все в порядке
   c3.gun(); // Все в порядке
}
```

8.7. Взаимодействие между const и immutable

Нередко квалификатор пытается подействовать на тип, который уже находится под влиянием другого квалификатора. Например:

```
struct A {
   const(int[]) c;
   immutable(int[]) i;
}

unittest {
   const(A) ca;
   immutable(A) ia;
}
```

Какие типы имеют поля ca.i и ia.c? Если бы квалификаторы применялись вслепую, получились бы типы const(immutable(int[])) и immutable(const(int[])) соответственно; очевидно, что-то тут лишнее, не говоря уже о типах ca.c и ia.i, когда один и тот же квалификатор применяется дважды!

При наложении одного квалификатора на другой D руководствуется простыми правилами композиции. Если квалификаторы идентичны, они сокращаются до одного. В противном случае как const(immutable(T)), так и immutable(const(T)) сокращаются до immutable(T), поскольку это более строгий тип. Эти правила применяются при распространении на типы элементов массива; например, элементы массива const(immutable(T)[]) имеют тип immutable(T), а не const(immutable(T)). При этом тип самого массива несократим.

8.8. Распространение квалификатора с параметра на результат

С и C++ определяют поверхностный квалификатор const с неприятной особенностью: функция, возвращающая параметр, должна либо повторять свое определение дважды – для константных и неконстантных данных, либо вести опасную игру. Показательный пример такой функции – функция strchr из стандартной библиотеки С, которая в ней определена так:

```
char* strchr(const char* input, int c);
```

Эта функция очищает типы от квалификаторов: несмотря на то что input – константное значение, которое, если рассуждать наивно, не должно измениться, возвращение в выводе указателя, порожденного от input, снимает с данных это обещание. strchr способствует появлению кода, изменяющего неизменяемые данные без приведения типов. C++ избавился от этой проблемы, введя два определения strchr:

```
char* strchr(char* input, int c);
const char* strchr(const char* input, int c);
```

Эти функции делают одно и то же, но их нельзя соединить в одну, поскольку в C++ нет средств, позволяющих сказать: «Если у аргумента есть квалификатор, пожалуйста, распространите его и на тип возвращаемого значения».

Для решения этой проблемы D предлагает «подстановочный» идентификатор квалификатора: inout. С участием inout объявление strchr выглядело бы так:

```
inout(char)* strchr(inout(char)* input, int c);
```

(Конечно же, в коде на D было бы предпочтительнее использовать массивы, а не указатели.) Компилятор понимает, что ключевое слово inout может быть заменено квалификатором immutable, const или ничем (последняя альтернатива имеет место в случае изменяемого входного значения). Он проверяет тело strchr, чтобы удостовериться в том, что код этой функции безопасно работает со всеми возможными типами входного значения.

Квалификатор может быть перенесен с метода на его результат, например:

```
class X {     }

class Y {
   private Y _another;
   inout(Y) another() inout {
      enforce(_another !is null);
```

```
        return _another;
    }
}
```

Метод another принимает объекты с любым квалификатором. Этот метод можно переопределить, что очень примечательно, поскольку inout можно воспринимать как обобщенный параметр, а обобщенные методы обычно переопределять нельзя. Компилятор способен сделать метод с inout переопределяемым, поскольку может проверить, работает ли код в теле этого метода со всеми квалификаторами.

8.9. Итоги

Квалификаторы типа выражают важные свойства типов, которые другие механизмы абстрагирования не позволяют выразить. Основное внимание в главе было уделено квалификатору типа immutable, предоставляющему очень надежные гарантии: неизменяемое значение за все время его жизни никогда не сможет быть изменено, транзитивно. Это очень полезное свойство. Оно позволяет обеспечить чисто функциональную семантику и помогает организовать безопасное разделение данных между потоками.

Сила квалификатора immutable также является его слабостью: он не допускает использование нескольких шаблонов обработки данных, когда за запись информации отвечают одни, а за ее чтение – другие. Эту проблему решает квалификатор const, выражающий контекстную неизменяемость: собственник значения const не может изменять данные, но другие части программы могут обладать этим правом.

Наконец, чтобы избежать повторения идентичного кода для функций, принимающих параметры с квалификатором и без квалификатора, был введен подстановочный квалификатор inout. Вместо ключевого слова inout подставляется immutable, const или пустое место при отсутствии квалификатора.

9

Обработка ошибок

Обработка ошибок – это слабо формализованная область программного инжиниринга, связанная с обработкой ожидаемых и возникших ошибочных ситуаций, способных помешать нормальному функционированию системы. Обработка исключительных ситуаций (исключений) – подход к обработке ошибок, принятый во многих современных языках (включая D) и уже успевший породить великое множество руководств, методик и, конечно, споров.

Исключения – это средство языка, реализующее обработку ошибок благодаря специальным обходным путям передачи управления. Если функции не удается вернуть вызвавшему ее коду осмысленный результат, она может породить объект исключения, в котором закодирована причина ошибки. Порождение исключений (throwing) – это карточка «Бесплатно освободитесь из тюрьмы»[1], освобождающая функцию от ее обычных обязанностей. Исключение пропускает всех инициаторов вызовов, не предусматривающих его обработку, и попадает в *место обработки*, где и принимаются чрезвычайные меры. В хорошо спроектированной программе гораздо меньше мест обработки, чем мест порождения исключений, что способствует централизованной обработке ошибок с многократным использованием кода. Все это было бы проблематично организовать на основе традиционных методик с вездесущими кодами ошибки.

9.1. Порождение и обработка исключительных ситуаций

D использует популярную модель исключений. Функция может инициировать исключения с помощью инструкции `throw` (см. раздел 3.11),

[1] Имеется в виду карточка из игры «Монополия». – *Прим. пер.*

порождающей объект особого типа. Чтобы завладеть этим объектом, код должен использовать инструкцию try (см. раздел 3.11), где этот объект указан в блоке catch. Перефразируя пословицу, лучше один пример кода, чем 1024 слова. Так что рассмотрим пример:

```
import std.stdio;

void main() {
   try {
      auto x = fun();
   } catch (Exception e) {
      writeln(e);
   }
}

int fun() {
   return gun() * 2;
}

int gun() {
   throw new Exception("Вернемся прямо в main");
}
```

Вместо того чтобы вернуть значение типа int, функция gun решает породить исключение, в данном случае это экземпляр класса Exception. В порожденном объекте можно передать любую информацию о том, что произошло. Управление передается исключительным путем (что освобождает от возвращения результата не только функцию, породившую исключение, но и всех инициаторов ее вызова) тому из инициаторов вызова, который готов обработать ошибку с помощью блока catch.

После выполнения инструкции throw функция fun полностью пропускается, поскольку она не готова обработать исключение. Вот в чем принципиальная разница между обработкой ошибок старой школы, когда ошибки вручную проводились через все уровни вызовов, и относительно новым подходом – обработкой исключений, когда управление искусно передается из места возникновения ошибки (gun) непосредственно туда, где есть все необходимое для ее обработки (блок catch в функции main). Такой подход обещает более простую, централизованную обработку ошибок, освобождая множество функций от обязанности проталкивать ошибки дальше по стеку; fun может оставаться в блаженном неведении о прямом сообщении между gun и main.

К сожалению, непосредственная передача потока управления из места порождения в место обработки – это также и слабое звено обработки исключений: на самом деле, то блаженное неведение – лишь несбыточная мечта. В действительности, функции, пересекаемые исключением, должны помнить о дополнительных неявных точках выхода и гарантировать поддержку инвариантов программы независимо от того, каким путем будет передано управление. D предоставляет надежные механиз-

мы, обеспечивающие сохранение инвариантности при возникновении исключений. В свое время мы обсудим их в этой главе.

9.2. Типы

Базовая иерархия исключений в D проста (рис. 9.1). Инструкция `throw` порождает не просто какие-то значения, а только объекты-потомки класса `Throwable`. В подавляющем большинстве случаев код действительно порождает исключение как экземпляр потомка класса `Exception`, подкласса `Throwable`. Это обычные исключения, после которых возможно восстановление, и они распознаются языком именно так. Исключения-наследники `Throwable`, но не `Exception` (такие как `AssertError`; см. главу 10) относятся к фатальным ошибкам, после которых восстановление невозможно, и должны использоваться в коде крайне редко, практически никогда. (О том, что язык гарантирует в случае фатальных ошибок, а что нет, см. подробнее в разделе 9.4.)

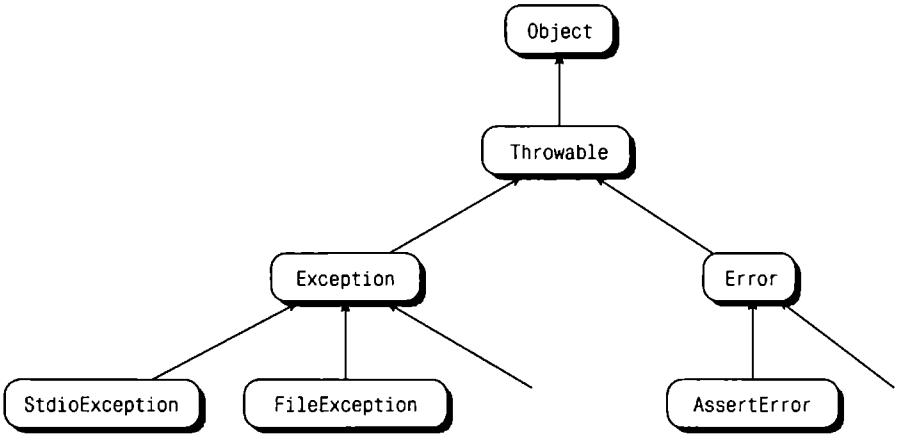

Рис. 9.1. Обычные исключения – потомки класса `Exception`, поэтому их можно обработать с помощью блока `catch(Exception)`. Класс `Error` – прямой наследник класса `Throwable`. Обычный код должен перехватывать только исключения типа `Exception` и потомков `Exception`. Остальные исключения лишь позволяют аккуратно завершить работу программы в случае нахождения ошибки в ее логике

Инструкция `try` может определять больше одного блока `catch`, например:

```
try {
   ..
} catch (SomeException e) {

} catch (SomeOtherException e) {

}
```

Исключения распространяются от места порождения до самого раннего места обработки, следуя правилу *первого совпадения*: сразу же после обнаружения блока catch, обрабатывающего исключение порожденного класса или его предка, этот блок catch активируется и порожденное исключение передается в него. Вот пример, порождающий и обрабатывающий исключения двух различных типов:

```
import std.stdio;

class MyException : Exception {
   this(string s) { super(s); }
}

void fun(int x) {
   if (x == 1) {
      throw new MyException("");
   } else {
      throw new StdioException("");
   }
}

void main() {
   foreach (i; 1    3) {
      try {
         fun(i);
      } catch (StdioException e) {
         writeln("StdioException");
      } catch (Exception e) {
         writeln("Exception");
      }
   }
}
```

Эта программа выводит на экран:

```
Exception
StdioException
```

Первый вызов fun порождает объект исключения типа MyException. При его сопоставлении с первым catch-обработчиком совпадения нет, но зато оно обнаруживается при сопоставлении со вторым блоком catch, поскольку MyException является потомком Exception. А в случае исключения, порожденного второй инструкцией throw, совпадение обнаруживается при сопоставлении с первым же catch-обработчиком. До первого совпадения этот процесс может затронуть и несколько уровней функций, как показывает следующий более замысловатый пример:

```
import std.stdio;

class MyException : Exception {
   this(string s) { super(s); }
}
```

```
void fun(int x) {
   if (x == 1) {
      throw new MyException("");
   } else if (x == 2) {
      throw new StdioException("");
   } else {
      throw new Exception("");
   }
}

void funDriver(int x) {
   try {
      fun(x);
   }
   catch (MyException e) {
      writeln("MyException");
   }
}

unittest {
   foreach (i; 1    4) {
      try {
         funDriver(i);
      } catch (StdioException e) {
         writeln("StdioException");
      } catch (Exception e) {
         writeln("Просто Exception");
      }
   }
}
```

Эта программа выводит на экран:

```
MyException
StdioException
Просто Exception
```

поскольку обработчики в соответствии с концепцией пробуются по мере того, как поток управления всплывает вверх по стеку вызовов.

У правила первого совпадения есть очевидный недостаток. Если блок `catch` **для исключения типа** `E1` **расположен до блока** `catch` **для исключения типа** `E2` **и при этом** `E2` **является подтипом** `E1`**, то обработчик для** `E2` **оказывается недоступным. В такой ситуации компилятор диагностирует ошибку. Например:**

```
import std.stdio;

void fun() {
   try {
```

```
    } catch (Exception e) {

    } catch (StdioException e) {
          // Ошибка!
          // Недоступный обработчик catch!
    }
}
```

Ошибочность подобного кода очевидна, но всегда есть угроза динамического маскирования между разными функциями. Любая функция легко может сделать недееспособными catch-обработчики вызвавшей ее функции. Тем не менее в большинстве случаев это не ошибка, а лишь нормальное следствие динамики стека вызовов.

9.3. Блоки finally

Инструкция try может завершаться блоком finally, что фактически означает: «Непременно выполните этот код, даже если наступит конец света или начнется потоп». Было или нет порождено исключение, блок finally будет выполнен просто как часть инструкции try, и вам решать, закончится ли это сбоем программы, порождением исключения, возвратом с помощью инструкции return или досрочным выходом из включающего цикла с помощью инструкции break. Например:

```
import std.stdio;

string fun(int x) {
   string result;
   try {
      if (x == 1) {
         throw new Exception("некоторое исключение");
      }
      result = "исключение не было порождено";
      return result;
   } catch (Exception e) {
      if (x == 2) throw e;
      result = "исключение было порождено и обработано: " ~ e.toString;
      return result;
   } finally {
      writeln("Выход из fun");
   }
}
```

При нормальном ходе выполнения функция fun вернет некоторое значение, при исключительном — породит исключение, но в любом случае она всегда напечатает в стандартный поток вывода: Выход из fun.

9.4. Функции, не порождающие исключения (nothrow), и особая природа класса Throwable

С помощью ключевого слова nothrow можно объявить функцию, не порождающую исключения:

```
nothrow int iDontThrow(int a, int b) {
   return a / b;
}
```

Функции, не порождающие исключения, уже упоминались в разделе 5.11.2. А вот и новый поворот сюжета: атрибут nothrow обещает, что функция не породит объект типа Exception. Но у функции по-прежнему остается право порождать объекты более грозного класса Throwable. Собственно, восстановление после исключений типа Throwable считается невозможным, поэтому компилятору разрешается не «продумывать» ход событий при возникновении такого исключения, соответственно он оптимизирует код исходя из предположения, что никакого исключения нет. Для функций с атрибутом nothrow компилятор упрощает последовательности входа и выхода, не планируя экстренные мероприятия на случай, если будет порождено исключение.

Проясним и подчеркнем особый статус класса Throwable. Первое правило для исключений Throwable: исключения Throwable не обрабатывают. Решив обработать такие исключения с помощью блока catch, вы не можете рассчитывать на то, что деструкторы структур будут вызваны, а блоки finally – выполнены. Это означает, что состояние вашей системы неопределенно и может быть нарушен целый ряд высокоуровневых инвариантов, на которые вы полагаетесь при нормальном исполнении. Тем не менее D гарантирует безопасность базовых типов и целостность стандартной библиотеки. Вы не можете рассчитывать на высокоуровневую целостность состояния своего приложения, поскольку не сработало неизвестное количество кода, обеспечивающего эту целостность. Так что при обработке Throwable вам доступны только несколько простых операций. Все, что вы сможете сделать в большинстве случаев, – это вывести сообщение об ошибке в стандартный поток или в файл журнала, попытаться сохранить в отдельном файле то, что еще можно сохранить, и, стиснув зубы, достойно завершить выполнение программы насколько это возможно.

9.5. Вторичные исключения

Иногда во время обработки исключения порождается еще одно. Например:

```
import std.conv;

class MyException : Exception {
```

```
    this(string s) { super(s); }
}

void fun() {
    try {
        throw new Exception("порождено в fun");
    } finally {
        gun(100);
    }
}

void gun(int x) {
    try {
        throw new MyException(text("порождено в gun #"  x));
    } finally {
        if (x > 1) {
            gun(x - 1);
        }
    }
}
```

Что происходит, когда вызвана функция fun? Ситуация на грани непредсказуемости. Во-первых, fun пытается породить исключение, но благодаря упомянутой привилегии блока finally всегда выполняться «даже если наступит конец света или начнется потоп», gun(100) вызывается тогда же, когда из fun вылетает Exception. В свою очередь, вызов gun(100) создает исключение типа MyException с сообщением "порождено в gun #100". Назовем второе исключение *вторичным*, чтобы отличать его от порожденного первым, которое мы назовем *первичным*. Затем уже функция gun с помощью блока finally порождает добавочные вторичные исключения – ровно 100 исключений. Такой код испугал бы и самого Макиавелли.

Ввиду необходимости обрабатывать вторичные исключения язык может выбрать один из следующих вариантов поведения:

- немедленно прервать выполнение;
- продолжить распространять первичное исключение, игнорируя все вторичные;
- заменить первичное исключение вторичным и продолжить распространять его;
- продолжить в той или иной форме распространять и первичное, и все вторичные исключения.

С точки зрения сохранения информации о происходящем последний подход видится наиболее обоснованным, но и самым сложным в реализации. Например, осмысленно обработать залп исключений гораздо труднее, чем одно исключение.

D выбрал подход простой и эффективный. Всякий объект типа Throwable содержит ссылку на следующий вторичный объект типа Throwable. Этот

вторичный объект доступен через свойство Throwable.next. Если вторичных исключений (больше) нет, значением свойства Throwable.next будет null. По сути, создается односвязный список с полной информацией обо всех вторичных ошибках в порядке их возникновения. В голове списка находится первичное исключение. Вот ключевые моменты определения Throwable:

```
class Throwable {
   this(string s);
   override string toString();
   @property Throwable next();
}
```

Разделение исключений на первичное и вторичные позволяет реализовать очень простую модель поведения. В любой момент, когда бы ни порождалось исключение, первичное исключение или уже порождено, или нет. Если нет, то порождаемое исключение становится первичным. Иначе порождаемое исключение добавляется в конец односвязного списка, головой которого является первичное исключение. Продолжив начатый выше пример, напечатаем всю цепочку исключений:

```
unittest {
   try {
      fun();
   } catch (Exception e) {
      writeln("Первичное исключение: исключение типа " ~ typeid(e).toString() ~ ": " ~ e.msg);
      Throwable secondary;
      while ((secondary = secondary.next) !is null) {
         writeln("Вторичное исключение: исключение типа " ~ typeid(e).toString() ~ ": " ~ e.msg);
      }
   }
}
```

Этот код напечатает:

```
Первичное исключение: исключение типа Exception порождено
в fun
Вторичное исключение: исключение типа MyException порождено
в gun #100
Вторичное исключение: исключение типа MyException порождено
в gun #99

Вторичное исключение: исключение типа MyException порождено
в gun #1
```

Вторичные исключения появляются в этой последовательности, поскольку присоединение к списку исключений выполняется в момент порождения исключения. Каждый раз инструкция throw извлекает первичное исключение (если есть), регистрирует новое исключение и инициирует или продолжает процесс порождения исключений.

Благодаря вторичным исключениям код на D может порождать исключения внутри деструкторов и блоков инструкций scope. В месте обработки исключения доступна полная информация о том, что произошло.

9.6. Раскрутка стека и код, защищенный от исключений

Пока исключение в полете, управление переносится из изначального места возникновения исключения через всю иерархию вверх до обработчика, при сопоставлении с которым происходит совпадение. Все участвующие в цепочке вызовов функции пропускаются. Ну, или почти все. Должную очистку после вызова пропускаемых функций обеспечивает так называемая *раскрутка стека* (*stack unwinding*) – часть процесса распространения исключения. Язык гарантирует, что пока исключение в полете, выполняются следующие фрагменты кода:

- деструкторы расположенных в стеке объектов-структур всех пропущенных функций;
- блоки finally всех пропускаемых функций;
- инструкции scope(exit) и scope(failure), действующие на момент порождения исключения.

Раскрутка стека – бесценная помощь в обеспечении корректности программы при возникновении исключений. Программы, использующие исключения, обычно предрасположены к утечке ресурсов. Многие ресурсы рассчитаны только на использование в режиме «получить/освободить», а при порождении исключений то и дело возникают малозаметные потоки управления, «забывающие» освободить ресурсы. Такие ресурсы лучше всего инкапсулировать в структуры, которые надлежащим образом освобождают управляемые ресурсы в своих деструкторах. Эта тема уже обсуждалась в разделе 7.1.3.6, пример такой инкапсуляции – стандартный тип File из модуля std.stdio. Структура File управляет системным дескриптором файла и гарантирует, что при уничтожении объекта типа File внутренний дескриптор будет корректно закрыт. Объект типа File можно копировать; счетчик ссылок отслеживает все активные копии; копия, уничтожаемая последней, закрывает файл в его низкоуровневом представлении. Этот популярный идиоматический подход к использованию деструкторов высоко ценят программисты на C++. (Данная идиома известна как RAII, см. раздел 6.16.) Другие языки и фреймворки также используют подсчет ссылок, вручную или автоматически.

Утечка ресурсов – лишь одно из проявлений более масштабной проблемы. Иногда шаблон «выполнить/отменить» связан с ресурсом, который невозможно «пощупать». Например, при записи текста HTML-файла многие теги (например,) полагается закрывать парным тегом (). Нелинейный поток управления, включающий порождение исключений,

может привести к генерации некорректно сформированных HTML-документов. Например:

```
void sendHTML(Connection conn) {
   conn.send("<html>");
       // Отправить полезную информацию в файл
   conn.send("</html>");
}
```

Если код между двумя вызовами conn.send преждевременно прервет выполнение функции sendHTML, то закрывающий тег не будет отправлен и результатом станет некорректный поток HTML-данных. Такую же проблему могла бы вызвать инструкция return, расположенная в середине sendHTML, но return можно хотя бы увидеть невооруженным глазом, просто внимательно просмотрев тело функции. Исключение же, напротив, может быть порождено любой из функций, вызывающих sendHTML (напрямую или косвенно). Из-за этого оценка корректности sendHTML становится гораздо более сложным и трудоемким процессом. Более того, у рассматриваемого кода есть серьезные проблемы со связанностью, поскольку корректность sendHTML зависит от того, как поведет себя при порождении исключений потенциально огромное число других функций.

Одно из возможных решений – имитировать RAII (даже если никакие ресурсы не задействованы): определить структуру, которая отсылает закрывающий тег в своем деструкторе. В лучшем случае это паллиатив. На самом деле, нужно гарантировать выполнение определенного кода, а не захламлять программу типами и объектами.

Другое возможное решение – воспользоваться блоком finally:

```
void sendHTML(Connection conn) {
   try {
      conn.send("<html>");

   } finally {
      conn.send("</html>");
   }
}
```

У этого подхода другой недостаток – масштабируемость, вернее ее отсутствие. Слабая масштабируемость finally становится очевидной, как только появляются несколько вложенных пар try/finally. Например, добавим в пример еще и отправку корректно закрытого тега <body>. Для этого потребуются *два* вложенных блока try/finally:

```
void sendHTML(Connection conn) {
   try {
      conn.send("<html>");
          // Отправить заголовок
      try {
         conn.send("<body>");
             // Отправить содержимое
```

```
        } finally {
            conn.send("</body>");
        }
    } finally {
        conn.send("</html>");
    }
}
```

Тот же результат можно получить альтернативным способом – с единственным блоком finally и дополнительной переменной состояния, отслеживающей, насколько продвинулось выполнение функции:

```
void sendHTML(Connection conn) {
    int step = 0;
    try {
        conn.send("<html>");
            // Отправить заголовок
        step = 1;
        conn.send("<body>");
        ... // Отправить содержимое
        step = 2;
    } finally {
        if (step > 1) conn.send("</body>");
        if (step > 0) conn.send("</html>");
    }
}
```

При таком подходе дела обстоят куда лучше, но теперь целый кусок кода посвящен только управлению состоянием, что затуманивает истинное предназначение функции.

Такие ситуации удобнее всего обрабатывать с помощью инструкций scope. Работающая функция в какой-то момент исполнения может включать инструкцию scope. Таким образом, любые фрагменты кода, представляющие собой логические пары, окажутся еще и объединенными физически.

```
void sendHTML(Connection conn) {
    conn.send("<html>");
    scope(exit) conn.send("</html>");
     .. // Отправить заголовок
    conn.send("<body>");
    scope(exit) conn.send("</body>");
        // Отправить содержимое
}
```

Новая организация кода обладает целым рядом привлекательных качеств. Во-первых, код теперь расположен линейно, без излишних вложенностей. Это позволяет легко разместить в коде сразу несколько пар типа «открыть/закрыть». Во-вторых, этот подход устраняет необходимость пристального рассмотрения кода функции sendHTML и вызываемых ею функций на предмет скрытых потоков управления, возникающих

при возможном порождении исключений. В-третьих, взаимосвязанные понятия сгруппированы, что упрощает чтение и сопровождение кода. В-четвертых, код получается компактным, поскольку накладные расходы на запись инструкции `scope` малы.

9.7. Неперехваченные исключения

Если найти обработчик для исключения не удалось, встроенный обработчик просто выводит сообщение об исключении в стандартный поток ошибок и завершает выполнение с ненулевым кодом выхода. Эта схема работает не только для исключений, распространяемых из `main`, но и для исключений, порождаемых блоками `static this`.

Как уже говорилось, обычно исключения типа `Throwable` не обрабатывают. В очень редких случаях вы, возможно, все же захотите обработать `Throwable` и принять какие-то экстренные меры, даже если наступит конец света или начнется потоп. Но при этом не рассчитывайте на осмысленное состояние системы в целом; логика вашей программы, скорее всего, будет нарушена, так что вы уже мало что сможете сделать.

10

Контрактное программирование

В мире, где мы все чаще доверяем свою жизнь разнообразным компьютерным системам, большим и малым, крайне важно обеспечивать корректность программ. Эта глава посвящена механизмам обеспечения корректности, вступающим в силу во время исполнения (а не во время компиляции, как контроль типов и другие семантические проверки, следящие за выполнением определенных требований корректности). Проверка корректности программ во время исполнения отчасти связана с обработкой ошибок, но не стоит смешивать два этих понятия. Точнее, за общей фразой «когда что-то идет не так» скрываются целых три пересекающихся, но отличных друг от друга области:

- *Обработка ошибок* (тема главы 9) имеет дело с методами и идиомами, помогающими справиться с ожидаемыми ошибками времени исполнения.
- *Техника обеспечения надежности* анализирует способность всей системы (например, программно-аппаратного комплекса) функционировать в соответствии со спецификацией. (В этой книге техника обеспечения надежности не рассматривается.)
- *Корректность программ* – это исследовательская область языка программирования, его статические и динамические средства, позволяющие доказать, что программа точно соответствует заданной спецификации. Системы типов – лишь наиболее известное средство, применяемое при доказательстве корректности программ (настоятельно рекомендуется прочесть увлекательную монографию «Доказательства – это программы» Уодлера [59]). В этой главе обсуждается контрактное программирование – парадигма обеспечения корректности программ.

Основное отличие корректности программ и обработки ошибок состоит в том, что вторая рассматривает ошибки *в пределах спецификации про-*

граммы (например, поврежденный файл данных или некорректный ввод данных пользователем), а первая – ошибки программирования, выводящие поведение программы *за пределы спецификации* (например, неверный расчет процентного значения, так что оно не попадает в интервал от 0 до 100, или неожиданно полученный отрицательный день недели в объекте типа `Date`). Пренебрежение этим важным отличием приводит к непростительным, но, к сожалению, все еще типичным промахам, вроде проверки файла или входных данных, переданных по сети, с помощью инструкции `assert`.

Контрактное программирование – это подход к определению компонентов программного обеспечения, предложенный Парнасом [45], а затем популяризированный Мейером [40] в языке программирования Eiffel. К настоящему моменту контрактное программирование выросло в популярную парадигму разработки программного обеспечения. Большинство основных языков программирования не ориентированы на поддержку контрактного программирования, но многие учреждения используют стандарты и соглашения, обеспечивающие его фундаментальные принципы. Контракты также являются областью активных исследований, включая такие непростые темы, как контракты для функций высокого порядка [24] и статическая верификация контрактов [61]. На данный момент D придерживается более простой, традиционной модели контрактного программирования, которую мы и рассмотрим в этой главе.

10.1. Контракты

Чтобы прояснить определение и процесс верификации модульных интерфейсов, контрактное программирование использует метафору из реальной жизни. Это метафора *заключения контракта*: когда сторона A (физическое или юридическое лицо) обязуется оказать стороне B определенную услугу, контракт между A и B описывает, что ожидается от B в обмен на эту услугу и что именно обязуется предоставить A, если B выполнит свою часть контракта.

Аналогично парадигма контрактного программирования определяет спецификацию функции как контракт между функцией (поставщик услуги) и инициатором ее вызова (клиент). Одна часть контракта регламентирует требования к инициатору, которые он должен выполнить, чтобы иметь право вызвать функцию. Другая часть контракта регламентирует обязательства функции по возвращаемому значению и/или побочным эффектам.

Ключевые понятия парадигмы контрактного программирования:

- *Утверждение* (*assertion*) – это не привязанная к конкретной функции проверка времени исполнения: проверяется некоторое условие, задаваемое с помощью инструкции `if`. Если условие ненулевое, инструкция `assert` не влияет на выполнение программы. В противном

случае assert порождает исключение типа AssertError. После исключения AssertError восстановление невозможно: этот класс не является потомком класса Exception, он прямой потомок класса Error, то есть исключения AssertError не следует обрабатывать.

- *Предусловие* (*precondition*) функции – это сумма условий, которые клиент должен выполнить, чтобы инициировать ее вызов. Условия могут относиться непосредственно к месту вызова (например, значения параметров) или к состоянию системы (например, доступность памяти).
- *Постусловие* (*postcondition*) функции – это сумма обязательств функции по нормальному возврату при условии удовлетворения предусловия.
- *Инвариант* (*invariant*) – это состояние, остающееся неизменным на протяжении цепочки вычислений. В языке D под инвариантами всегда понимается состояние объекта до и после вызова метода.

Контрактное программирование изящно обобщает ряд проверенных временем понятий, которые мы сегодня воспринимаем как данность. Например, сигнатура функции – это настоящий контракт. Рассмотрим функцию из модуля std.math стандартной библиотеки:

```
double sqrt(double x);
```

Сама сигнатура – заключение контракта: инициатор должен предоставить ровно одно значение типа double, а функция в свою очередь – вернуть одно значение типа double. Нельзя сделать вызов sqrt("hello") или присвоить результат вызова sqrt строке. Еще интереснее, что *можно* сделать вызов sqrt(2), даже если 2 – значение типа int, а не double: сигнатура дает компилятору достаточно информации, чтобы тот мог привести значение 2 к типу double и тем самым помочь клиенту выполнить требования, предъявляемые к входным данным. Функция может обладать побочными эффектами, а если их нет, этот факт можно отразить с помощью атрибута pure:

```
// Никаких побочных эффектов
pure double sqrt(double x);
```

Это гораздо более строгий, более обязывающий контракт с точки зрения sqrt, поскольку он запрещает sqrt иметь побочные эффекты. Наконец, атрибут nothrow позволяет заключить еще более детальный (и ограничивающий) контракт:

```
// Никаких побочных эффектов, никаких исключений
// (именно такое обьявление находится в модуле std.math)
pure nothrow double sqrt(double x);
```

Теперь мы точно знаем, что эта функция или вернет значение типа double, или приведет к останову программы, или инициирует бесконечный цикл. Кроме этого она абсолютно ничего сделать не может. Итак, мы уже вовсю пользуемся контрактами, просто записывая сигнатуры.

Чтобы вы ощутили контрактную мощь сигнатур функций, приведем одно небольшое историческое свидетельство. Ранняя, еще до появления стандарта версия языка C (названная «K&R C» в честь своих создателей Кернигана и Ричи) была с сюрпризом: при вызове необъявленной функции компилятор K&R C считал, что она обладает следующей сигнатурой:

```
// Если вы не объявили функцию sqrt, но вызвали ее,
// то это равносильно тому, что вы объявили ее следующим образом,
// а потом вызвали
int sqrt(.  );
```

Другими словами, если вы забывали включить с помощью директивы `#include` заголовочный файл `math.h` (предоставляющий корректную сигнатуру для `sqrt`), то *могли* без помех со стороны компилятора сделать вызов `sqrt("hello")`. (Многоточием обозначено переменное число аргументов, одно из самых небезопасных средств C.)

Коварство ошибки заключалось в том, что вызов `sqrt(2)`, скомпилированный с файлом `math.h` и без этого файла, делал совершенно разные вещи. С директивой `#include` компилятор перед вызовом `sqrt` конвертировал аргумент 2 в `2.0`, а без директивы между сторонами возникало трагическое непонимание: инициатор отправлял 2, а `sqrt` считывала бинарное представление этого значения так, будто это было число с плавающей запятой, что в 32-разрядном формате IEEE составляет `2.8026e-45`. Язык C осознал серьезность этой проблемы и устранил ее, введя требование для всех функций предоставлять прототипы.

Простые контракты для функций можно формулировать с помощью их атрибутов и типов. Количество атрибутов ограничено, но новый тип легко определить в любой момент. Насколько важны типы в описании контрактов? Ответ, к сожалению, таков (по крайней мере, с текущей технологией): типы не являются адекватным средством для выражения даже умеренно сложных контрактов.

Разработчик мог бы изложить контракт функции в документации по этой функции, но, уверен, все согласятся, что такой подход далеко не удовлетворителен. Пользователи компонента не всегда читают документацию с должным вниманием, а если и читают, то могут просто ошибиться. Кроме того, документация обычно отстает от проекта и реализации, особенно когда спецификации нетривиальны и постоянно изменяются (как нередко бывает).

Контрактное программирование избрало более простой подход: контрактные требования представляют собой исполняемые предикаты – фрагменты кода, описывающие контракт в виде логических условий. Рассмотрим по очереди каждый из этих предикатов.

10.2. Утверждения

Выражение assert было определено в разделе 2.3.4.1, и с тех пор мы использовали его повсеместно, по умолчанию признавая полезность понятия «утверждение». В дополнение отметим, что большинство языков включают некоторый механизм проверки утверждений в виде примитива языка или библиотечной конструкции.

Напомним, что выражение с ключевым словом assert – это утверждение: *по плану* такое выражение должно быть истинным (ненулевым) *при любом запуске программы и независимо от входных данных*:

```
int a, b;

assert(a == b);
assert(a == b, "a и b разные");
```

Утверждаемое выражение обычно является логическим, но может иметь любой тип, значение которого можно проверить с помощью конструкции if: числовой тип, массив, указатель или ссылка на класс. Если выражение нулевое, при выполнении инструкции assert порождается исключение типа AssertError; в противном случае ничего не происходит. Если порождено исключение класса AssertError, необязательный строковый параметр становится частью этого объекта. Строка вычисляется, только если утверждаемое выражение действительно нулевое, что позволяет сэкономить на некоторых потенциально затратных вычислениях:

```
import std.conv;

void fun() {
   int a, b;
   ...
   assert(a == b);
   assert(a == b, text(a, " и ", b, " разные"));
}
```

Функция std.conv.text конвертирует и объединяет все свои аргументы в строку. Чтобы выполнить эти операции, нужно потрудиться: выделить память, провести преобразования и т. д. Было бы расточительно выполнять всю эту работу в случае успешного выполнения инструкции assert, так что второй аргумент вычисляется, только если первый оказывается нулевым.

Что должна делать инструкция assert в случае неудачи? Возможный вариант (именно так и поступает одноименный макрос из C) – принудительно завершить выполнение программы. А в языке D инструкция assert порождает в таком случае исключение. Тем не менее это не обычное исключение; это объект класса AssertError, дочернего класса Error – сверхисключения, о котором шла речь в разделе 9.2.

Объект типа AssertError, порожденный инструкцией assert, проходит через обработчики catch(Exception), как горячий нож сквозь масло. Это хорошо, поскольку неудачи при проверке утверждений свидетельствуют о логических ошибках в вашей программе, и обычно хочется, чтобы логические ошибки как можно скорее и аккуратнее останавливали выполнение приложения.

Чтобы перехватить исключение типа AssertError, укажите в обработчике catch в качестве аргумента не класс Exception или его потомка, а класс Error или прямо AssertError. Но повторяю: вряд ли вам когда-либо пригодится перехват исключений типа Error.

10.3. Предусловия

Предусловия – это контрактные обязательства, которые должны быть выполнены при входе в функцию. Предположим, мы хотим с помощью контракта потребовать для функции fun неотрицательные входные данные. Это предусловие, которое функция fun предъявляет вызывающему ее коду. В языке D предусловие записывается так:

```
double fun(double x)
in {
   assert(x >= 0);
}
body {
   // Реализация fun

}
```

Контракт in автоматически выполняется перед выполнением тела функции. Фактически это более простая версия кода:

```
double fun(double x) {
   assert(x >= 0);
   // Реализация fun

}
```

Но мы еще увидим, как важно отделить предусловие от тела функции, особенно при использовании объектов и наследования.

Некоторые языки сводят ограничения к логическим выражениям и автоматически порождают исключение, если такое логическое выражение ложно, например:

```
// Не D
double fun(double x)
in (x >= 0)
body {

}
```

D более гибок – он позволяет проверять даже те предусловия, которые трудно отобразить одиночным логическим выражением. Кроме того, он предоставляет вам право порождать исключения любого типа, а не только AssertError. Например, функция fun может породить исключение, запоминающее ошибочные входные данные:

```
import std.conv, std.exception;

class CustomException : Exception {
   private string origin;
   private double val;
   this(string msg, string origin, double val) {
      super(msg);
      this.origin = origin;
      this.val = val;
   }
   override string toString() {
      return text(origin, ": "  super.toString(), val);
   }
}

double fun(double x)
in {
   if (x !>= 0) {
      throw new CustomException(
         "Отрицательное значение входного параметра"
         "fun"  x);
   }
}
body {
   double y;
   // Реализация fun

   return y;
}
```

Но не злоупотребляйте этой гибкостью. Как уже говорилось, инструкция assert порождает исключение типа AssertError, которое не является обычным исключением. Сигнализировать о невыполнении предусловия лучше всего с помощью исключений типа AssertError и других потомков класса Error, а не Exception, потому что невыполнение предусловия свидетельствует о серьезной логической ошибке в вашей программе, а такие ошибки не планируется обрабатывать обычным способом.

На самом деле, чтобы запретить такое некорректное использование контрактов, компилятор предпринимает определенные меры. Во-первых, внутри блока in нельзя выполнить инструкцию return, а значит, вам не разрешат полностью пропустить тело функции с помощью контракта. Во-вторых, D строго запрещает изменять параметры внутри контракта. Например, следующий код ошибочен:

```
double fun(double x)
in {
   if (x <= 0) x = 0; // Ошибка!
   // Нельзя изменить параметр x внутри контракта!
}
body {
   double y;

   return y;
}
```

Но хотя компилятор и мог бы настаивать на функциональной чистоте контракта (что было бы логично), он этого не делает. То есть внутри контракта вы по-прежнему можете изменять глобальные переменные или генерировать вывод. Эта свобода была предоставлена не просто так: «нечистые» контракты полезны во время сеансов отладки, и запретить их было бы слишком жестоко. Контрактный код предназначен лишь для того, чтобы проверять подчинение контракту и порождать исключение в случае нарушения контракта, – и больше ни для чего.

10.4. Постусловия

Имея лишь in-контракт, функция fun остается антисимметричной и как бы нечестной: она предъявляет инициатору вызова требования, но не предоставляет никаких гарантий. С какой стати тогда инициатору вызова трудиться, передавая в fun неотрицательное число? Для проверки постусловий служит out-контракт. Предположим, fun гарантирует, что вернет результат в диапазоне от 0 до 1:

```
double fun(double x)
// Как и раньше
in {
   assert(x >= 0);
}
// добавлено
out(result) {
   assert(result >= 0 && result <= 1);
}
body {
   // Реализация fun
   double y;

   return y;
}
```

Если in-контракт тела функции породит исключение, блок out вообще не будет выполнен. Постусловие выполняется, только если предусловие выполнено и тело функции без проблем вернуло результат. Параметр result, передаваемый в блок out, содержит значение, которое функция готова вернуть. Передача параметра result не является необходимой;

out{...} – тоже корректный out-контракт, который не использует результат либо применяется к функции типа void. В нашем примере в качестве result выступает копия y.

Как и in-контракт, out-контракт лишь проверяет, но не изменяет. Взаимодействие out-контракта с внешним миром сводится к отсутствию действий (если постусловие выполнено) или к порождению исключения (если постусловие не выполнено). Отметим, что out-контракт – не лучшее место для наведения порядка в последний момент. Вычисляйте результат в теле функции и проверяйте его в блоке out. Следующий код не компилируется по двум причинам: out-контракт пытается изменить переменную result, а также делает (безвредную, но подозрительную) попытку изменить аргумент:

```
int fun(int x)
out(result) {
   x = 42;                        // Ошибка!
      // Нельзя изменить параметр x внутри контракта!
   if (result < 0) result = 0; // Ошибка!
      // Нельзя изменить результат внутри контракта!
}
body {
   ...
}
```

10.5. Инварианты

Инвариант – условие, которое остается истинным на определенных этапах вычисления. Например, чистая функция гарантирует, что состояние программы останется полностью неизменным от начала и до конца выполнения функции. Такая гарантия очень надежна, но слишком строга, чтобы к ней можно было часто прибегать.

Более узконаправленный инвариант может касаться индивидуального объекта, и именно с такой моделью работает D. Рассмотрим, например, простой класс Date, который хранит значения дня, месяца и года в виде отдельных целых чисел:

```
class Date {
   private uint year, month, day;

}
```

Разумно предположить, что в течение всего времени жизни объекта типа Date его члены day, month и year не должны принимать бессмысленные значения. Выразить такое требование можно с помощью инварианта:

```
import std.algorithm, std.range;

class Date {
private:
```

```
    uint year, month, day;
    invariant() {
       assert(1 <= month && month <= 12);
       switch (day) {
          case 29:
             assert(month != 2 || leapYear(year));
             break;
          case 30:
             assert(month != 2);
             break;
          case 31:
             assert(longMonth(month));
             break;
          default:
             assert(1 <= day && day <= 28);
             break;
       }
       // Никаких ограничений на год
    }
    // Вспомогательные функциии
    static pure bool leapYear(uint y) {
       return (y % 4) == 0 && (y % 100 || (y % 400) == 0);
    }
    static pure bool longMonth(uint m) {
       return !(m & 1) == (m > 7);
    }
public:

}
```

С помощью трех проверок для чисел месяца 29, 30 и 31 выполняется обработка особых случаев для февраля високосного года. Проверяющая функция `longMonth` возвращает `true`, если в месяце 31 день, и работает по принципу «месяц с четным номером является длинным тогда и только тогда, когда наступает после июля», что соответствует истине (длинные месяцы имеют номера 1, 3, 5, 7, 8, 10 и 12).

Инвариант должен выполняться всегда, для всех корректных объектов типа `Date`. Теоретически компилятор может генерировать проверки инвариантов в какой угодно момент. Он мог бы, например, своей властью добавить проверку инварианта после каждой инструкции, что было бы не только неэффективно, но и некорректно. Рассмотрим инициализацию одного объекта типа `Date` другим:

```
// Внутри класса Date
void copy(Date another) {
   year = another.year;
   __call_invariant(); // Вставлено компилятором
   month = another.month;
   __call_invariant(); // Вставлено компилятором
   day = another.day;
```

```
    __call_invariant(); // Вставлено компилятором
}
```

Вполне возможно, что где-то между этими инструкциями состояние Date временно становится некорректным, так что вставка вычисления инварианта после каждой инструкции – прием тоже некорректный. (Например, в ходе присвоения дате, в текущий момент содержащей 29 февраля 2012 г., даты 1 августа 2015 г. объект временно переводится в состояние 29 февраля 2015, а эта дата некорректна.)

А если вставлять вызовы инварианта в начале и в конце каждого метода? Снова не то. Предположим, что функция переводит дату на месяц вперед. Такая функция могла бы, например, отслеживать ежемесячные события. Функция должна уделять внимание лишь корректировке дня ближе к концу месяца, так чтобы дата изменялась, например с 31 августа на 30 сентября.

```
// Внутри класса Date
void nextMonth() {
   __call_invariant();              // Вставлено компилятором
   scope(exit) __call_invariant(); // Вставлено компилятором
   if (month == 12) {
      ++year;
      month = 1;
   } else {
      ++month;
      adjustDay();
   }
}
// Вспомогательная функция
private void adjustDay() {
   __call_invariant();              // Вставлено компилятором
                                    // (ПРОБЛЕМАТИЧНО)
   scope(exit) __call_invariant(); // Вставлено компилятором
                                    // (ПРОБЛЕМАТИЧНО)
   switch (day) {
      case 29:
         if (month == 2 && !leapYear(year)) day = 28;
         break;
      case 30:
         if (month == 2) day = 28 + leapYear(year);
         break;
      case 31:
         if (month == 2) day = 28 + leapYear(year);
         else if (!isLongMonth(month)) day = 30;
         break;
      default:
         // Ничего не делать
         break;
   }
}
```

Функция nextMonth заботится о смене лет и использует вспомогательную локальную (private) функцию adjustDay, чтобы обеспечить корректность даты. Здесь-то и кроется проблема: на входе в adjustDay инвариант может оказаться «сломанным»! Разумеется, может – ведь функция adjustDay предназначена именно для *исправления* объекта класса Date!

Что делает функцию adjustDay особенной? Уровень доступа: это локальная (закрытая) функция, доступная только другим функциям, имеющим право модифицировать объект класса Date. В общем случае на входе в закрытую функцию и на выходе из нее допускается несоблюдение инварианта. Где инвариант точно должен выполняться, так это на границах общедоступных методов: объект не допустит, чтобы клиентские операции застигли или оставили его в некорректном состоянии.

Как насчет защищенных (protected) функций? В соответствии с обсуждением в разделе 6.7.6 уровень доступа protected лишь немногим лучше public. Тем не менее посчитали, что требовать соблюдения инварианта на границах защищенных функций – чересчур строго.

Если класс определяет инвариант, компилятор автоматически вставляет обращения к этому инварианту в следующих местах:

1. В *конце* всех конструкторов.
2. В *начале* деструктора.
3. В *начале* и *конце* всех общедоступных нестатических методов.

Представим, что мы надели рентгеновские очки, позволяющие видеть код, вставленный компилятором в класс Date. Вот что мы бы увидели:

```
class Date {
   private uint day, month, year;
   invariant() { ... }
   this(uint day, uint month, uint year) {
      scope(exit) __call_invariant();

   }
   ~this() {
      __call_invariant();

   }
   void somePublicMethod() {
      __call_invariant();
      scope(exit) __call_invariant();

   }
}
```

Добавим пару слов о конструкторе и деструкторе. Вспомните, что говорилось о жизненном цикле объекта в разделе 6.3: после выделения памяти под объект он считается полностью дееспособным. Следовательно, даже если конструктор порождает исключение, он все равно должен оставить объект в состоянии, подчиняющемся условию инварианта.

10.6. Пропуск проверок контрактов. Итоговые сборки

Контракты предназначены только для проверки внутренней логики приложения. В соответствии с этим соглашением большинство (если не все) программных систем, поддерживающих контракты, также предоставляют режим, в котором проверка контрактов игнорируется. Предположительно этот режим применяется только к досконально осмотренным, проверенным и протестированным программам.

Любой компилятор языка D предоставляет флаг (`-release` в случае эталонной реализации), при задании которого контракты полностью игнорируются, то есть компилятор выполняет синтаксический анализ и проверки типов для всего кода контрактов, но не оставляет от него и следа в исполняемом бинарном файле. Результат итоговой сборки исполняется без проверки контрактов (что рискованнее), но зато на полной скорости. Если в приложении все разложено по полочкам, дополнительный риск, связанный с пропуском проверок контрактов, очень мал и полностью искупается увеличением скорости. Возможность запуска без контрактов – веский довод в пользу предупреждения о том, что код *не* должен использовать контракты для обычных проверок, которые вполне могут завершаться неудачей. Контракты должны быть зарезервированы для недопустимых ошибок, отражающих изъян логики вашего приложения. Повторяю: никогда не проверяйте корректность ввода данных пользователем с помощью контрактов. Кроме того, вспомните неоднократные предупреждения о том, что внутри `assert`, `in` и `out` нельзя выполнять никакие важные действия. Теперь совершенно ясно почему: программа, которая так нехорошо поступает, почему-то по-разному ведет себя в промежуточной и итоговой сборках.

Одна из наиболее частых ошибок – утверждение выражений с побочными эффектами, например `assert(++x < y)`, которому суждено стать настоящей головоломкой. Это худшее из возможного: ошибка появляется только в итоговой сборке, когда у вас по определению меньше средств для обнаружения источника проблемы.

10.6.1. enforce – это не (совсем) assert

Жаль, что удобные инструкции с ключевым словом `assert` исчезают из итоговых сборок. Вместо

```
if (!expr1) throw new SomeException;

if (!expr2) throw new SomeException;

if (!expr3) throw new SomeException;
```

можно было бы написать просто

```
assert(expr1);

assert(expr2);

assert(expr3);
```

Ввиду лаконичности инструкции assert множество библиотек предоставляют «assert с гарантией» – средство, которое проверяет условие и в случае нулевого результата порождает исключение независимо от того, как вы провели компиляцию – в режиме итоговой сборки или нет. Такие «контролеры» есть в C – это VERIFY, ASSERT_ALWAYS и ENFORCE. Язык D определяет аналогичную функцию enforce в модуле std.exception. Используйте enforce с тем же синтаксисом, что и assert:

```
enforce(expr1);
enforce(expr2, "Это не совсем верно");
```

Если выражение-аргумент – нулевое, функция enforce порождает исключение типа Exception независимо от того, как вы скомпилировали программу – в режиме итоговой или промежуточной сборки. Порождение исключения другого типа задается так:

```
import std.exception;
bool something = true;

enforce(something, new Error("Что-то не так"));
```

Если значение something нулевое, порождается объект, переданный во втором аргументе; функция enforce использует механизм ленивых аргументов[1], так что если значение выражения something ненулевое, никакого создания объекта не произойдет.

Несмотря на то что assert и enforce выглядят и ведут себя сходным образом, служат они принципиально разным целям. Не забывайте о различиях между этими двумя конструкциями:

- assert проверяет логику вашего приложения, а enforce – условия возникновения ошибок, не угрожающих целостности вашего приложения;
- assert порождает только исключение типа AssertError, после которого восстановление невозможно, а enforce по умолчанию порождает исключение, после которого восстановление возможно (и может породить любое исключение – достаточно указать его во втором аргументе);
- assert может исчезнуть, поэтому, пытаясь выяснить логику потока управления в своей функции, не стоит обращать внимание на утверждения; enforce никогда не исчезает, так что после вызова enforce(e) можно предполагать, что значение e ненулевое.

[1] Ленивые аргументы описаны в разделе 5.2.4. – *Прим. науч. ред.*

10.6.2. assert(false) – останов программы

Если во время компиляции известно, что константа равна нулю, то утверждение с этой константой (вида `assert(false)`, `assert(0)` и `assert(null)`) ведет себя несколько иначе, чем обычное утверждение.

В режиме промежуточной сборки инструкция `assert(false);` не делает ничего особенного: она лишь порождает исключение типа `AssertError`.

Зато в режиме итоговой сборки инструкция `assert(false);` не исключается при компиляции; она всегда вызывает останов программы. Но в этом случае не будет ни исключения, ни шанса продолжить выполнение после того, как очередь дошла до `assert(false)`. Произойдет программный *сбой*. На машинах марки Intel для этого есть инструкция `HLT` (от halt – стой!), принудительно завершающая выполнение программы.

Многие из нас воспринимают сбой как опасное событие, свидетельствующее о том, что программа вышла из-под контроля. Это мнение широко распространено, скорее всего потому, что выполнение программ, реально вышедших из-под контроля, обычно завершается сбоем. Однако `assert(false)` – это весьма контролируемый способ остановить выполнение программы. На самом деле, в некоторых операционных системах `HLT` автоматически загружает ваш отладчик, позиционируя его на той самой инструкции `assert`, которая вызвала сбой.

Для чего нужно это особое поведение `assert(false)`? Самое очевидное применение касается программ системного уровня. Необходим переносимый способ выполнить `HLT`, а `assert(false)` хорошо вписывается в язык. Добавим, что компилятор в курсе семантики `assert(false)`, например он запрещает оставлять после выражения `assert(false)` «мертвый» код:

```
int fun(int x) {
   ++x;
   assert(false);
   return x; // Ошибка!
             // Инструкция недоступна!
}
```

В других ситуациях, наоборот, `assert(false)` поможет пресечь ошибку компилятора. Рассмотрим, например, вызов только что упомянутой стандартной функции `std.exception.enforce(false)`:

```
import std.exception;

string fun() {
   ..
   enforce(false, "продолжать невозможно"); // Всегда порождает исключение
   assert(false);                           // Эта инструкция недоступна
}
```

Вызов `enforce(false)` всегда порождает исключение, но компилятор не знает об этом. Инструкция `assert(false);` дает компилятору понять, что эта точка недостижима. Завершить выполнение `fun` можно и с помощью

инструкции return "";, но если позже кто-нибудь закомментирует вызов enforce, fun начнет возвращать фиктивные значения. Выражение assert(false) – настоящий *deus ex machina*, избавляющий ваш код от таких ситуаций.

10.7. Контракты – не для очистки входных данных

Этот раздел посвящен спорному вопросу в связи с контрактами, послужившему источником длительных дебатов. Суть вопроса: «Куда лучше поместить выполняемую функцией проверку: в контракт или в тело функции?»

При первом знакомстве с контрактным программированием многие поддаются искушению перенести большинство проверок в контракты. Рассмотрим пример функции readText, которая полностью считывает текст из файла в строку. Вооружившись контрактами, можно определить эту функцию так:

```
import std.file, std.utf;

string readText(in char[] filename)
out(result) {
   std.utf.validate(result);
}
body {
   return cast(string) read(filename);
}
```

(На самом деле, readText – это функция из стандартной библиотеки; найти ее можно в модуле std.file.)

Функция readText полагается на две другие функции для работы с файлом. Во-первых, чтобы целиком загрузить файл в буфер памяти, readText вызывает функцию read. Буфер памяти имеет тип void[]; функция readText преобразует это значение в строку с помощью оператора cast. Но останавливаться на этом нельзя: что если файл содержит некорректные UTF-знаки? Чтобы проверить результат преобразования, out-контракт вызывает применительно к результату readText функцию std.utf.validate, которая порождает исключение типа UtfException, если буфер содержит некорректный UTF-знак.

Все было бы хорошо, если бы не один основополагающий момент: контракты должны подтверждать состоятельность *логики* приложения, а не корректность его входных данных. То, что не считается проблемой самого приложения, не должно включаться в контракт. Кроме того, контракты не предназначены для изменения семантики приложения – отсюда и намеренные ограничения D на то, что можно изменить внутри контракта.

Если предположить, что все контракты успешно выполняются, поведение и результат работы приложения не должны зависеть от того, что

контракты действительно делают или не делают. Это очень простой и легко запоминающийся тест, помогающий определить, что является контрактом, а что нет. Контракты – это проверки спецификации, и если в корректной реализации их убрать, это не повлияет на дальнейшую работу реализации! Вот как должны работать контракты. Возможно, позитивное мироощущение заставляет вас надеяться, что файл всегда будет правильным, но соответствующее требование не вписывается в спецификацию. Вот корректное определение `readText`, делающее проверку неотъемлемой частью функции:

```
import std.file, std.utf;

string readText(in char[] filename) {
   auto result = cast(string) read(filename);
   std.utf.validate(result);
   return result;
}
```

Все это приводит к такому ответу на вопрос о месте расположения проверок: если проверка касается логики приложения, то ее следует поместить в контракт; в противном случае проверку помещают в тело функции и никогда не пропускают.

Звучит неплохо, но как определить понятие «логика приложения» для приложения, построенного из отдельных стандартных библиотек, написанных независимыми сторонами? Представьте большую библиотеку общего назначения, такую как Microsoft Windows API или K Desktop Environment. Подобные API используются множеством приложений, и библиотечные функции неизбежно получают аргументы, не соответствующие спецификации. (На самом деле, API операционной системы должен быть *рассчитан* на получение всевозможных некорректных аргументов.) Если приложение не выполнило предусловие вызова библиотечной функции, чья это вина? Очевидно, что это ошибка приложения, но именно библиотека получает удар – в виде нестабильности, непредсказуемого поведения, испорченного внутреннего состояния библиотеки, сбоев, всех этих неприятностей сразу. Хоть это и явная несправедливость, но, к сожалению, достается за эти проблемы в основном библиотеке («Библиотека Xyz склонна к нестабильности и неожиданным причудам»), а не кривой логике приложений, которые ее используют.

Широко распространенный API общего назначения должен проверять входные данные всех своих функций как положено – не в контрактах. Отсутствие проверки аргумента – однозначно ошибка библиотеки. Ни один пресс-секретарь не станет размахивать книгой или статьей со словами: «Мы везде использовали контрактное программирование, так что это не наша вина».

Разве это помешало бы указывать в предусловии те же диапазоны аргументов функций? Вовсе нет. Все зависит от определения и разграничения «логики приложения» и «пользовательского ввода». С точки зрения

функции как неотъемлемой части приложения получение аргументов – часть логики приложения. А для функции общего назначения из независимо поставляемой библиотеки аргументы – не что иное, как пользовательский ввод.

С другой стороны, для библиотеки совершенно естественно использовать контракты в своих локальных функциях. Эти функции относятся к внутреннему функционированию библиотеки и недоступны пользовательскому коду, так что вполне разумно позволить им использовать контракты для выражения строгого соответствия спецификации.

10.8. Наследование

Часто цитируемый принцип подстановки Барбары Лисков [38] гласит, что наследование – это возможность подстановки: экземпляр производного класса (потомка) можно подставить везде, где ожидается экземпляр его базового класса (предка). Такое понимание, по сути, определяет взаимодействие контрактов с наследованием.

В реальном мире взаимоотношение между контрактами и возможностью подстановки таково: по заключении контракта к подменяющему подрядчику предъявляются следующие требования – он должен обладать *не меньшей* квалификацией для выполнения указанной работы, выполнить эту работу с допустимым отклонением *не больше* указанного в контракте и требовать вознаграждение *не больше* указанного в контракте. Здесь есть некоторая гибкость, но она не допускает ужесточения предусловий контракта или ослабления его постусловий. Если что-либо из этого произойдет, контракт станет недействительным и его придется переписать. Гибкость касается лишь изменений, не ухудшающих соглашение контракта: подменяющему разрешается требовать *меньше*, а предлагать *больше*.

10.8.1. Наследование и предусловия

Вернемся к примеру с классом `Date`. Допустим, мы определили очень простой, легковесный класс `BasicDate`, который предоставляет лишь минимальную функциональность, а реализацию усовершенствований оставляет классам-потомкам. В `BasicDate` есть функция `format`, которая принимает аргумент типа `string` (спецификация формата) и возвращает строку с датой, отформатированную заданным образом:

```
import std.conv;

class BasicDate {
   private uint day, month, year;
   string format(string spec)
   in {
      // Требование равенства spec строке "%Y/%m/%d"
      assert(spec == "%Y/%m/%d");
```

```
    }
    body {
        // Упрощенная реализация
        return text(year, '/' month, '/', day);
    }
    ...
}
```

Контракт, заключенный функцией `Date.format`, требует, чтобы спецификация формата точно соответствовала `"%Y/%m/%d"`, то есть «год в четырехзначном формате, затем косая черта, затем месяц, затем косая черта, затем день». Это единственный формат, о поддержке которого заботится `BasicDate`. Классы-потомки могут добавить локализацию, интернационализацию и все, что только можно.

Наследник класса `BasicDate` класс `Date` желает предложить лучший примитив формата, например, позволяющий спецификаторам `%Y`, `%m` и `%d` занимать любые позиции и перемешиваться с произвольными знаками. Кроме того, должно быть разрешено сочетание знаков `%%`, поскольку оно представляет сам знак `%`. Повторное вхождение одних и тех же спецификаторов также должно быть разрешено. Чтобы воплотить все это в жизнь, `Date` пишет собственный контракт:

```
import std.regex;

class Date : BasicDate {
    override string format(string spec)
    in {
        auto pattern = regex(`^(%[mdY%]|[^%])*$`);
        assert(!match(spec, pattern).empty);
    }
    body {
        string result;
        ...
        return result;
    }

}
```

`Date` накладывает свои ограничения на `spec` с помощью регулярного выражения. Регулярные выражения — это бесценная помощь в манипуляции строками: классический труд Фридла «Регулярные выражения» [26] не просто рекомендуется к прочтению, а горячо рекомендуется. Не углубляясь в регулярные выражения, достаточно сказать, что `"^(%[mdY%]|[^%])*$"` означает: «строка, которая может быть пустой или содержать повторяющуюся сколько угодно раз следующую комбинацию знаков: `%`, а за ним любой знак из `m`, `d`, `Y` и `%` либо любой знак, отличный от `%`». Эквивалентный код, проводящий сопоставление такому шаблону «вручную», оказался бы гораздо более многословным. Утверждение гарантирует, что при сопоставлении строки и шаблона совпадений будет больше нуля, то есть то, что сопоставление сработает. (Более подробно

применение регулярных выражений в D описано в онлайн-документации по стандартному модулю std.regex.)

Каков *совокупный* контракт Date.format? Он должен учитывать контракт BasicDate.format, но в то же время ослаблять его. Вполне приемлемо, если in-контракт соблюден не будет, но при этом обязательно должен выполняться контракт потомка. Кроме того, контракт Date.format ни при каких обстоятельствах не должен ужесточать контракт BasicDate.format. Появляется правило: в переопределенном методе сначала выполнить контракт предка – если выполнение завершится удачей, выполнить тело функции; в противном случае выполнить контракт потомка – в случае успеха выполнить тело функции, иначе сообщить о неудаче.

Другими словами, in-контракты комбинируются с использованием дизъюнкции в сочетании с коротким замыканием: точно один контракт должен выполниться, и контракт предка пробуется первым. Таким образом, исключается вероятность того, что контракт потомка труднее удовлетворить, чем контракт предка. С другой стороны, потомок предоставляет второй шанс пройти тест предусловия, не пройденный в первый раз.

Это правило прекрасно работает для Date и BasicDate. Сначала составной контракт проверяет входные данные на соответствие шаблону "%Y/%m/%d" В случае успеха форматирование продолжается. Иначе выполняется проверка на соответствие контракту класса-потомка. В случае удачного исхода этого теста форматирование снова может быть продолжено.

Код, сгенерированный для комбинированного контракта, выглядит так:

```
void __in_contract_Date_format(string spec) {
   try {
      // Попробовать контракт предка
      this.BasicDate.__in_contract_format(spec);
   } catch (Throwable) {
      // Контракт предка не выполнен, попробовать контракт потомка
      this.Date.__in_contract_format(spec);
   }
   // Успех, можно выполнить тело функции
}
```

10.8.2. Наследование и постусловия

С out-контрактами дело обстоит ровно наоборот. При замене предка потомком переопределенная функция должна предлагать *больше*, чем обещано в контракте. Так что гарантии, предоставляемые out-контрактом метода предка, переопределяющий метод всегда должен предоставлять во всей полноте (в отличие от случая с in-контрактом).

С другой стороны, это означает, что класс-предок должен заключать максимально свободный контракт, не рискуя чересчур ограничить класс-потомок. Например, потребовав от возвращаемой строки соответствия формату «год/месяц/день», метод BasicDate.format полностью запретил

бы использование любым классом-потомком любого другого формата. BasicDate.format мог бы обязать своих потомков выполнять не столь строгий контракт – например, если строка формата не пуста, то использовать пустую строку в качестве выходных данных запрещено:

```
import std.range, std.string;

class BasicDate {
   private uint day, month, year;
   string format(string spec)
   out(result) {
      assert(!result.empty || spec.empty);
   }
   body {
      return std.string.format("%04s/%02s/%02s"  year, month, day);
   }
    ..
}
```

Класс Date устанавливает планку немного выше: он вычисляет ожидаемую длину результата по спецификации формата, а затем сравнивает длину действительного результата с ожидаемой длиной:

```
import std.algorithm, std.regex;

class Date : BasicDate {
   override string format(string spec)
   out(result) {
      bool escaping;
      size_t expectedLength;
      foreach (c; spec) {
         switch (c) {
            case '%':
               if (escaping) {
                  ++expectedLength;
                  escaping = false;
               } else {
                  escaping = true;
               }
               break;
            case 'Y':
               if (escaping) {
                  expectedLength += 4;
                  escaping = false;
               }
               break;
            case 'm': case 'd':
               if (escaping) {
                  expectedLength += 2;
                  escaping = false;
               }
```

```
                break;
            default:
                assert(!escaping);
                ++expectedLength;
                break;
            }
        }
        assert(walkLength(result) == expectedLength);
    }
    body {
        string result;

        return result;
    }

}
```

(Почему `walkLength(result)` вместо `result.length`? Потому что количество знаков в строке в кодировке UTF может быть меньше, чем ее длина в кодовых единицах.) Даны два контракта. Каким должен быть комбинированный `out`-контракт? Ответ прост: контракт класса-предка также должен быть проверен. Далее, если класс-потомок обещает выполнить *дополнительные* контрактные обязательства, они также должны быть соблюдены. Это простая конъюнкция. Следующий код представляет собой то, что должен сгенерировать компилятор, чтобы соединить контракты базового и производного классов:

```
void __out_contract_Date_format(string spec) {
    this.BasicDate.__out_contract_format(spec);
    this.Date.__out_contract_format(spec);
    // Успех
}
```

10.8.3. Наследование и инварианты

Как и в случае `out`-контрактов, мы имеем дело с конъюнкцией, отношением «И»: помимо собственного инварианта класс должен следить за соблюдением инвариантов всех своих предков. Для класса не существует способа ослабить инвариант своего предка. Текущая версия компилятора делает вызовы блоков `invariant` сверху донизу по иерархии, но для того, кто реализует `invariant`, порядок не важен, ведь инварианты не должны обладать побочными эффектами.

10.9. Контракты и интерфейсы

Возможно, наиболее интересное применение контрактов – в сочетании с интерфейсами. Интерфейс – это один сложный контракт. С такой трактовкой хорошо согласуется то, что каждый из методов интерфейса должен описывать абстрактный контракт – контракт без тела. Кон-

тракт описывается в терминах еще не реализованных примитивов, определенных интерфейсом.

Предположим, что требуется усовершенствовать интерфейс Stack, определенный в разделе 6.14. Приведем его для справки:

```
interface Stack(T) {
   @property bool empty();
   @property ref T top();
   void push(T value);
   void pop();
}
```

Присоединим к интерфейсу контракты, раскрывающие принципы взаимодействия между его примитивами. Контракты интерфейса выглядят точно так же, как обычные контракты, только у них нет тела.

```
interface Stack(T) {
   @property bool empty();
   @property ref T top()
   in {
      assert(!empty);
   }
   void push(T value)

   out {
      assert(value == top);
   }
   void pop()
   in {
      assert(!empty);
   }
}
```

В конце метода интерфейса с контрактом больше не требуется ставить точку с запятой. С новым определением интерфейса Stack его реализации будут вынуждены работать в рамках ограничений, определенных контрактами этого интерфейса. Положительным моментом является то, что усовершенствованный с помощью контрактов интерфейс Stack – хорошая спецификация стека, которую программисту одновременно легко читать и динамически проверять.

Как говорилось в разделе 10.7, во время компиляции контракты интерфейса Stack могут быть опущены. Если вы пожелаете определить библиотеку с контейнером для повсеместного многоцелевого использования, возможно, полезно будет считать вызовы методов входными данными от пользователя. В таком случае более подходящей может оказаться идиома NVI (см. раздел 6.9.1). Интерфейс стека, использующий NVI с целью всегда проверять, корректны ли вызовы, выглядел бы так:

```
interface NVIStack(T) {
protected:
   ref T topImpl();
```

```
    void pushImpl(T value);
    void popImpl();
  public:
    @property bool empty();

    final @property ref T top() {
      enforce(!empty);
      return topImpl();
    }

    final void push(T value) {

      pushImpl(value);
      enforce(value == topImpl());
    }

    final void pop() {
      assert(!empty);
      popImpl();
    }
  }
```

NVIStack повсюду использует enforce-тест, который невозможно стереть во время компиляции, а также определяет методы push, pop и top как финальные, то есть запрещает реализациям их переопределять. Хорошо здесь то, что всю основную обработку ошибок можно переложить с каждой из реализаций на интерфейс – неплохой метод повторного использования кода и разделения ответственности. Реализации интерфейса NVIStack могут без опаски полагаться на то, что pushImpl, popImpl и topImpl всегда вызываются в корректных состояниях, и оптимизировать свои методы с учетом этого.

11

Расширение масштаба

Поговорка гласит, что программу в 100 строк можно заставить работать, даже если она нарушает все законы правильного кодирования. Эта поговорка расширяема: действительно, можно написать программу в 10000 строк, уделяя внимание лишь деталям кода и не соблюдая никаких более масштабных правил надлежащей модульной разработки. Возможно, где-то есть и проекты в несколько миллионов строк, нарушающие немало правил крупномасштабной разработки.

Многие твердые принципы разработки программного обеспечения также производят впечатление расширяемых. Разделение ответственности и сокрытие информации одинаково работают в случае небольшого модуля и при соединении целых приложений. Воплощение этих принципов, тем не менее, варьируется в зависимости от уровня, на котором эти принципы применяются. Эта часть посвящена сборке более крупных сущностей – целых файлов, каталогов, библиотек и программ.

Определяя свой подход к крупномасштабной модульности, D следует отдельным хорошо зарекомендовавшим себя принципам, а также вводит пару любопытных инноваций относительно поиска имен.

11.1. Пакеты и модули

Единицей компиляции, защиты и инкапсуляции является физический файл. Единицей логического объединения множества файлов является каталог. Вот и все сложности. С точки зрения модульности мы обращаемся к файлу с исходным кодом на D как к *модулю*, а к каталогу, содержащему файлы с исходным кодом на D, – как к *пакету*.

Нет причин думать, что исходному коду программы на самом деле будет удобнее в какой-нибудь супер-пупер базе данных. D использует «базу данных», которую долгое время настраивали лучшие из нас и которая

прекрасно интегрируется со средствами обеспечения безопасности, системой управления версиями, защитой на уровне ОС, журналированием – со всем, что бы вы ни назвали, а также устанавливает низкий барьер входа для широкомасштабной разработки, поскольку основные необходимые инструменты – это редактор и компилятор.

Модуль D – это текстовый файл с расширением `.d` или `.di`. Инструментарий D не учитывает расширения файлов при обработке, но по общему соглашению в файлах с расширением `.d` находится код реализации, а в файлах с расширением `.di` (от D interface – интерфейс на D) – код интерфейсов. Текст файла должен быть в одной из следующих кодировок: UTF-8, UTF-16, UTF-32. В соответствии с небольшим стандартизированным протоколом, известным как BOM (byte order mark – метка порядка байтов), порядок следования байтов в файле (в случае UTF-16 или UTF-32) определяется несколькими первыми байтами файла. В табл. 11.1 показано, как компиляторы D идентифицируют кодировку файлов с исходным кодом (в соответствии со стандартом Юникод [56, раздел 2]).

Таблица 11.1. Для различения файлов с исходным кодом на D используются метки порядка байтов. Шаблоны проверяются сверху вниз, первое же совпадение при сопоставлении устанавливает кодировку файла. xx – любое ненулевое значение байта

Если первые байты...	...то кодировка файла – ...	Игнорировать эти байты?
`00 00 FE FF`	UTF-32 с прямым порядком байтов[1]	✓
`FF FE 00 00`	UTF-32 с обратным порядком байтов[2]	✓
`FE FF`	UTF-16 с прямым порядком байтов	✓
`FF FE`	UTF-16 с обратным порядком байтов	✓
`00 00 00 xx`	UTF-32 с прямым порядком байтов	
`xx 00 00 00`	UTF-32 с обратным порядком байтов	
`00 xx`	UTF-16 с прямым порядком байтов	
`xx 00`	UTF-16 с обратным порядком байтов	
Что-то другое	UTF-8	

В некоторых файлах метка порядка байтов отсутствует, но у D есть средство, позволяющее автоматически недвусмысленно определить кодировку. Процедура автоопределения тонко использует тот факт, что любой правильно построенный модуль на D должен начинаться хотя бы с нескольких знаков, встречающихся в кодировке ASCII, то есть с кодовых точек Юникода со значением меньше 128. Ведь в соответствии

[1] Прямой порядок байтов – от старшего к младшему байту. – *Прим. пер.*

[2] Обратный порядок байтов – от младшего к старшему байту. – *Прим. пер.*

с грамматикой D правильно построенный модуль должен начинаться или с ключевого слова языка D (состоящего из знаков Юникода с ASCII-кодами), или с ASCII-пробела, или с комментария, который начинается с ASCII-знака /, или с пары директив, начинающихся с #, которые также должны состоять из ASCII-знаков. Если выполнить проверку на соответствие шаблонам из табл. 11.1, перебирая эти шаблоны сверху вниз, первое же совпадение недвусмысленно укажет кодировку. Если кодировка определена ошибочно, вреда от этого все равно не будет – файл, несомненно, и так ошибочен, поскольку начинается со знаков, которые не может содержать корректный код на D.

Если первые два знака (после метки порядка байтов, если она есть) – это знаки #!, то эти знаки плюс следующие за ними знаки вплоть до первого символа новой строки \n игнорируются. Это позволяет использовать средство «shebang»[1] тем системам, которые его поддерживают.

11.1.1. Объявления import

Для получения доступа к благам стандартной библиотеки в примерах кода из предыдущих глав обычно использовалась инструкция `import`:

```
import std.stdio; // Получить доступ к writeln и всему остальному
```

Чтобы включить один модуль в другой, укажите имя модуля в объявлении `import`. Имя модуля должно содержать путь до него относительно каталога, где выполняется компиляция. Рассмотрим пример иерархии каталогов (рис. 11.1).

Предположим, компиляция выполняется в каталоге `root`. Чтобы получить доступ к определениям файла `widget.d` из любого другого файла, этот другой файл должен содержать объявление верхнего уровня:

```
import widget;
```

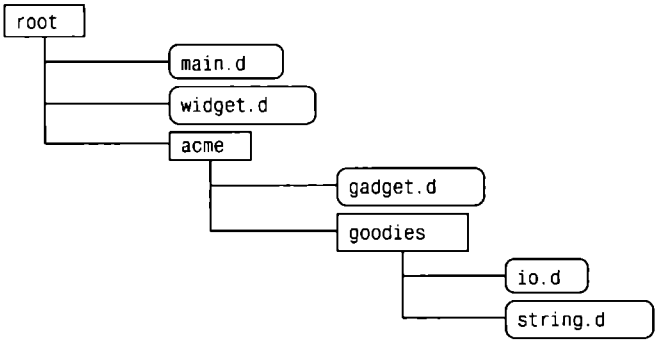

Рис. 11.1. Пример структуры каталога

[1] «Shebang» – от англ. *sharp-bang* или *hash-bang*, произношение символов #! – *Прим. науч. ред.*

«Объявление верхнего уровня» – это объявление вне всех контекстов (таких как функция, класс и структура)[1]. Встретив это объявление import, компилятор начнет искать widget.di (сначала) или widget.d (потом) начиная с каталога root, найдет widget.d и импортирует его идентификаторы. Чтобы использовать файл, расположенный глубже в иерархии каталогов, другой файл проекта должен содержать объявление import с указанием относительного пути до него от каталога root с точкой в качестве разделителя:

```
import acme.gadget;
import acme.goodies.io;
```

В объявлениях import мы обычно используем списки значений, разделенных запятыми. Два предыдущих объявления эквивалентны следующему:

```
import acme.gadget, acme.goodies.io;
```

Обратите внимание: файл, расположенный на более низком уровне иерархии каталогов, такой как gadget.d, *также* должен указывать путь к другим файлам относительно каталога root, где выполняется компиляция, а не относительно собственного расположения. Например, чтобы получить доступ к идентификаторам файла io.d, файл gadget.d должен содержать объявление:

```
import acme.goodies.io;
```

а не

```
import goodies.io;
```

Другой пример: если файл io.d хочет включить файл string.d, то он должен содержать объявление import acme.goodies.string, хотя оба этих файла находятся в одном каталоге. Разумеется, в данном случае предполагается, что компиляция выполняется в каталоге root. Если вы перешли в каталог acme и компилируете gadget.d *там*, он должен содержать объявление import goodies.io.

Порядок включения модулей не имеет значения. Язык задуман таким образом, что семантика модуля не зависит от порядка, в каком этот модуль включает другие модули.

Объявление import присоединяет только идентификаторы (символы), следовательно, пакеты и модули на D должны иметь имена, являющиеся допустимыми идентификаторами языка D (см. раздел 2.1). Например, если у вас есть файл 5th_element.d, то вы просто не сможете включить его в другой модуль, поскольку «5th_element» не является допустимым идентификатором D. Точно так же, если вы храните файлы в каталоге input-output, то не сможете использовать этот каталог как

[1] Текущие версии реализации позволяют включать модули на уровне классов и функций. – *Прим. науч. ред.*

пакет D. Иными словами, все файлы и каталоги, содержащие исходный код на языке D, должны носить лишь имена, являющиеся допустимыми идентификаторами. Дополнительное соглашение: имена всех пакетов и модулей не содержат заглавных букв. Цель этого соглашения – предотвратить путаницу в операционных системах с нестрогой обработкой регистра букв в именах файлов.

11.1.2. Базовые пути поиска модулей

Анализируя объявление `import`, компилятор выполняет поиск не только относительно текущего каталога, где происходит компиляция. Иначе невозможно было бы использовать ни одну из стандартных библиотек или других библиотек, развернутых за пределами каталога текущего проекта. В конце концов мы постоянно включаем модули из пакета `std`, хотя в поле зрения наших проектов нет никакого подкаталога `std`. Как же работает этот механизм?

Как и многие другие языки, D позволяет задать набор *базовых путей* (*roots*), откуда начинается поиск модулей. С помощью аргумента, передаваемого компилятору из командной строки, к списку базовых путей поиска модулей можно добавить любое количество каталогов. Точный синтаксис этой операции зависит от компилятора; эталонный компилятор `dmd` использует флаг командной строки `-I`, сразу за которым указывается путь, например `-Ic:\Programs\dmd\src\phobos` для Windows-версии и `-I/usr/local/src/phobos` для UNIX-версии. С помощью дополнительных флагов `-I` можно добавить любое количество путей в список путей поиска.

Например, при анализе объявления `import path.to.file` сначала подкаталог `path/to`[1] ищется в текущем каталоге. Если такой подкаталог существует, запрашивается файл `file.d`. Если файл найден, поиск завершается. В противном случае такой же поиск выполняется, начиная с каждого из базовых путей, заданных с помощью флага `-I`. Поиск завершается при первом нахождении модуля; если же все каталоги пройдены безрезультатно, компиляция прерывается с ошибкой «модуль не найден».

Если компонент `path.to` отсутствует, поиск модуля будет осуществляться непосредственно в базовых каталогах.

Для пользователя было бы обременительно добавлять флаг командной строки только для того, чтобы получить доступ к стандартной библиотеке или другим широко используемым библиотекам. Вот почему эталонный компилятор (и фактически любой другой) использует простой конфигурационный файл, содержащий несколько флагов командной строки по умолчанию, которые автоматически добавляются к каждому вызову компилятора из командной строки. Сразу после инсталляции

[1] В тексте / используется в качестве обобщенного разделителя; необходимо понимать, что реальный разделитель зависит от системы.

компилятора конфигурационный файл должен содержать такие установки, чтобы с его помощью можно было найти, по крайней мере, библиотеку поддержки времени исполнения и стандартную библиотеку. Поэтому если вы просто введете

```
% dmd main.d
```

то компилятор сможет найти все артефакты стандартной библиотеки, не требуя никаких параметров в командной строке. Чтобы точно узнать, где ищется каждый из модулей, можно при запуске компилятора `dmd` добавить флаг `-v` (от verbose – подробно). Подробное описание того, как установленная вами версия D загружает конфигурационные параметры, вы найдете в документации для нее (в случае `dmd` документация размещена в Интернете [18, 19, 20, 21]).

11.1.3. Поиск имен

Как ни странно, в D нет глобального контекста или глобального пространства имен. В частности, нет способа определить истинно глобальный объект, функцию или имя класса. Причина в том, что единственный способ определить такую сущность – разместить ее в модуле, а у любого модуля должно быть имя. В свою очередь, имя модуля порождает именованный контекст. Даже `Object`, предок всех классов, в действительности не является глобальным именем: на самом деле, это объект `object.Object`, поскольку он вводится в модуле `object`, поставляемом по умолчанию. Вот, например, содержимое файла `widget.d`:

```
// Содержимое файла widget.d
void fun(int x) {
    ..
}
```

С определением функции `fun` не вводится глобально доступный идентификатор `fun`. Вместо этого все, кто включает модуль `widget` (например, файл `main.d`), получают доступ к идентификатору `widget.fun`:

```
// Содержимое main.d
import widget;

void main() {
   widget.fun(10); // Все в порядке, ищем функцию fun в модуле widget
}
```

Все это очень хорошо и модульно, но при этом довольно многословно и неоправданно строго. Если нужна функция `fun` и никто больше ее не определяет, почему компилятор не может просто отдать предпочтение `widget.fun` как единственному претенденту?

На самом деле, именно так и работает поиск имен. Каждый включаемый модуль вносит свое пространство имен, но когда требуется найти идентификатор, предпринимаются следующие шаги:

1. Идентификатор ищется в текущем контексте. Если идентификатор найден, поиск успешно завершается.
2. Идентификатор ищется в контексте текущего модуля. Если идентификатор найден, поиск успешно завершается.
3. Идентификатор ищется во *всех* включенных модулях:
   - если идентификатор не удается найти, поиск завершается неудачей;
   - если идентификатор найден в единственном модуле, поиск успешно завершается;
   - если идентификатор найден более чем в одном модуле и этот идентификатор не является именем функции, поиск завершается с ошибкой, выводится сообщение о дублировании идентификатора;
   - если идентификатор найден более чем в одном модуле и этот идентификатор является именем функции, применяется механизм разрешения имен при кроссмодульной перегрузке (см. раздел 5.5.2).

Привлекательным следствием такого подхода является то, что клиентский код обычно может быть кратким, а многословным только тогда, когда это действительно необходимо. В предыдущем примере функция `main.d` могла вызвать `fun` проще, без каких-либо «украшений»:

```
// Содержимое main.d
import widget;

void main() {
   fun(10); // Все в порядке, идентификатор fun определен
            // только в модуле widget
}
```

Пусть в файле `io.d` также определена функция `fun` с похожей сигнатурой:

```
// Содержимое io.d из каталога acme/goodies
void fun(long n) {

}
```

И пусть модуль с функцией `main` включает и файл `widget.d`, и файл `io.d`. Тогда «неприукрашенный» вызов `fun` окажется ошибочным, но уточненные вызовы с указанием имени модуля по-прежнему будут работать нормально:

```
// Содержимое main.d
import widget, acme.goodies.io;

void main() {
   fun(10); // Ошибка!
            // Двусмысленный вызов функции fun():
            // идентификатор fun найден в модулях widget и acme.goodies.io
```

```
    widget.fun(10);          // Все в порядке, точное указание
    acme.goodies.io.fun(10); // Все в порядке, точное указание
}
```

Обратите внимание: сама собой двусмысленность не проявляется. Если вы не попытаетесь обратиться к идентификатору в двусмысленной форме, компилятор никогда не пожалуется.

11.1.3.1. Кроссмодульная перегрузка функций

В разделе 5.5.2 обсуждается вопрос перегрузки функций в случае их расположения в разных модулях и приводится пример, в котором модули, определяющие функцию с одним и тем же именем, вовсе не обязательно порождают двусмысленность. Теперь, когда мы уже знаем больше о модулях и модульности, пора поставить точку в этом разговоре.

Угон функций (*function hijacking*) представляет собой особенно хитрое нарушение модульности. Угон функций имеет место, когда функция в некотором модуле состязается за вызовы из функции в другом модуле и принимает их на себя. Типичное проявление угона функций: работающий модуль ведет себя по-разному в зависимости от того, каковы другие включенные модули, или от порядка, в котором эти модули включены.

Угоны могут появляться как следствие непредвиденных эффектов в других случаях исправно выполняемых и благонамеренных правил. В частности, кажется логичным, чтобы в предыдущем примере, где модуль `widget` определяет `fun(int)`, а модуль `acme.goodies.io` – `fun(long)`, вызов `fun(10)`, сделанный в `main`, был присужден функции `widget.fun`, поскольку это «лучший» вариант. Однако это один из тех случаев, когда лучшее – враг хорошего. Если модуль с функцией `main` включает только `acme.goodies.io`, то вызов `fun(10)`, естественно, отдается `acme.goodies.io.fun` как единственному кандидату. Однако если на сцену выйдет модуль `widget`, вызов `fun(10)` неожиданно переходит к `widget.fun`. На самом деле, `widget` вмешивается в контракт, который изначально заключался между `main` и `acme.goodies.io` – ужасное нарушение модульности.

Неудивительно, что языки программирования остерегаются угона. C++ разрешает угон функций, но большинство руководств по стилю программирования советуют этого приема избегать; а Python, как и многие другие языки, и вовсе запрещает любой угон. С другой стороны, переизбыток воздержания может привести к излишне строгим правилам, воспитывающим привычку использовать в именах длинные строки идентификаторов.

D разрешает проблему угона оригинальным способом. Основной руководящий принцип подхода D к кроссмодульной перегрузке состоит в том, что добавление или уничтожение включаемых модулей не должно влиять на разрешение имени функции. Возня с инструкциями `import` может привести к тому, что ранее компилируемые модули перестанут компилироваться, а ранее некомпилируемые модули станут компилируемыми. Опасный сценарий, который D исключает, – тот, при котором,

поиграв с объявлениями import, вы оставите программу компилируемой, но с разными результатами разрешения имен при перегрузке.

Для любого вызова функции, найденного в модуле, справедливо, что если имя этой функции найдено более чем в одном модуле *и* если вызов может сработать с версией функции из любого модуля, то такой вызов ошибочен. Если же вызов можно заставить работать лишь при одном варианте разрешения имени, такой вызов легален, поскольку при таких условиях нет угрозы угона.

В приведенном примере, где widget определяет fun(int), а acme.goodies.io — fun(long), положение дел в модуле main таково:

```
import widget, acme.goodies.io;

void main() {
   fun(10);   // Ошибка! Двусмысленная кроссмодульная перегрузка!
   fun(10L);  // Все в порядке, вызов недвусмысленно переходит
              // к acme.goodies.io.fun
   fun("10"); // Ошибка! Ничего не подходит!
}
```

Добавив или удалив из инструкции import идентификатор widget или acme.goodies.io, можно заставить сломанную программу работать, или сломать работающую программу, или оставить работающую программу работающей — но никогда с различными решениями относительно вызовов fun в последнем случае.

11.1.4. Объявления public import

По умолчанию поиск идентификаторов во включаемых модулях не является транзитивным. Рассмотрим каталог на рис. 11.1. Если модуль main включает модуль widget, а модуль widget в свою очередь включает модуль acme.gadget, то поиск идентификатора, начатый из main, в модуле acme.gadget производиться *не* будет. Какие бы модули ни включал модуль widget, это лишь деталь реализации модуля widget, и для main она не имеет значения.

Тем не менее может статься, что модуль widget окажется лишь расширением другого модуля или будет иметь смысл лишь в связке с другим модулем. Например, определения из модуля widget могут использовать и требовать так много определений из модуля acme.goodies.io, что для любого другого модуля было бы бесполезно использовать widget, не включив также и acme.goodies.io. В таких случаях вы можете помочь клиентскому коду, воспользовавшись объявлением public import:

```
// Содержимое widget.d
// Сделать идентификаторы из acme.goodies.io видимыми всем клиентам widget
public import acme.goodies.io;
```

Данное объявление public import делает все идентификаторы, определенные модулем acme/goodies/io.d, видимыми из модулей, включающих

widget.d (внимание) *как будто widget.d определил их сам.* По сути, public import добавляет в widget.d объявление alias для каждого идентификатора из io.d. (Дублирование кода объектов не происходит, только некоторое дублирование идентификаторов.) Предположим, что модуль io.d определяет функцию print(string), а в функцию main.d поместим следующий код:

```
import widget;

void main() {
   print("Здравствуй");         // Все в порядке, идентификатор print найден
   widget.print("Здравствуй"); // Все в порядке, widget фактически
                               // определяет print
}
```

Что если на самом деле включить в main и модуль acme.goodies.io? Попробуем это сделать:

```
import widget;
import acme.goodies.io; // Излишне, но безвредно

void main() {
   print("Здравствуй");                  // Все в порядке.
   widget.print("Здравствуй");           // ...в порядке.
   acme.goodies.io.print("Здравствуй"); //  .. и в порядке!
}
```

Модулю io.d вред не нанесен: тот факт, что модуль widget определяет псевдоним для acme.goodies.io, ни в коей мере не влияет на исходный идентификатор. Дополнительный псевдоним – это просто альтернативное средство получения доступа к одному и тому же определению.

Наконец, в некотором более старом коде можно увидеть объявления private import. Такая форма использования допустима и аналогична обычному объявлению import.

11.1.5. Объявления static import

Иногда добавление включаемого модуля в неявный список для поиска идентификаторов при объявлении import (в соответствии с алгоритмом из раздела 11.1.3) может быть нежелательным. Бывает уместным желание осуществлять доступ к определенному в модуле функционалу только с явным указанием полного имени (а-ля имямодуля.имяидентификатора, а не имяидентификатора).

Простейший случай, когда такое решение оправданно, – использование очень популярного модуля в связке с модулем узкого назначения при совпадении ряда идентификаторов в этих модулях. Например, в стандартном модуле std.string определены широко используемые функции для обработки строк. Если вы взаимодействуете с устаревшей системой, применяющей другую кодировку (например, двухбайтный набор знаков, известный как DBCS – Double Byte Character Set), то захотите

использовать идентификаторы из std.string в большинстве случаев, а идентификаторы из собственного модуля dbcs_string – лишь изредка и с точным указанием. Для этого нужно просто указать в объявлении import для dbcs_string ключевое слово static:

```
import std.string;          // Определяет функцию string toupper(string)
static import dbcs_string; // Тоже определяет функцию string toupper(string)

void main() {
   auto s1 = toupper("hello");              // Все в порядке
   auto s2 = dbcs_string.toupper("hello"); // Все в порядке
}
```

Уточним: если бы этот код не включал объявление import std.string, первый вызов просто не компилировался бы. Для static import поиск идентификаторов не автоматизируется, даже когда идентификатор недвусмысленно разрешается.

Бывают и другие ситуации, когда конструкция static import может быть полезной. Сдержать автоматический поиск и использовать более многословный, но одновременно и более точный подход может пожелать и модуль, включающий множество других модулей. В таких случаях ключевое слово static полезно использовать с целыми списками значений, разделенных запятыми:

```
static import teleport, time_travel, warp;
```

Или располагать его перед контекстом, заключенным в скобки, с тем же результатом:

```
static {
   import teleport;
   import time_travel, warp;
}
```

11.1.6. Избирательные включения

Другой эффективный способ справиться с конфликтующими идентификаторами – включить лишь определенные идентификаторы из модуля. Для этого используйте следующий синтаксис:

```
// Содержимое main.d
import widget : fun, gun;
```

Избирательные включения обладают точностью хирургического лазера: данное объявление import вводит ровно *два* идентификатора – fun и gun. После избирательного включения невидим даже идентификатор widget! Предположим, модуль widget определяет идентификаторы fun, gun и hun. В таком случае fun и gun можно будет использовать только так, будто их определил сам модуль main. Любые другие попытки, такие как hun, widget.hun и даже widget.fun, незаконны:

```
// Содержимое main.d
import widget : fun, gun;

void main() {
   fun();        // Все в порядке
   gun();        // Все в порядке
   hun();        // Ошибка!
   widget.fun(); // Ошибка!
   widget.hun(); // Ошибка!
}
```

Высокая точность и контроль, предоставляемые избирательным включением, сделали это средство довольно популярным – есть программисты, не приемлющие ничего, кроме избирательного включения; особенно много таких среди тех, кто прежде работал с языками, обладающими более слабыми механизмами включения и управления видимостью. И все же необходимо отметить, что другие упомянутые выше механизмы уничтожения двусмысленности, которые предоставляет D, ничуть не менее эффективны. Полный контроль над включаемыми идентификаторами был бы гораздо более полезен, если бы механизм поиска идентификаторов, используемый D по умолчанию, не был безошибочным.

11.1.7. Включения с переименованием

Большие проекты имеют тенденцию создавать запутанные иерархии пакетов. Чрезмерно ветвистые структуры каталогов – довольно частый артефакт разработки, особенно в проектах, где заранее вводят щедрую, всеобъемлющую схему именования, способную сохранить стабильность даже при непредвиденных добавлениях в проект. Вот почему нередки ситуации, когда модулю приходится использовать очень глубоко вложенный модуль:

```
import util.container.finite.linear.list;
```

В таких случаях может быть весьма полезно *включение с переименованием*, позволяющее присвоить сущности `util.container.finite.linear.list` короткое имя:

```
import list = util.container.finite.linear.list;
```

С таким объявлением `import` программа может использовать идентификатор `list.symbol` вместо чересчур длинного идентификатора `util.container.finite.linear.list.symbol`. Если исходить из того, что модуль, о котором идет речь, определяет класс `List`, в итоге получим:

```
import list = util.container.finite.linear.list;

void main() {
   auto lst1 = new list.List;                              // Все в порядке
   auto lst2 = new util.container.finite.linear.list.List; // Ошибка!
      // Идентификатор util не определен!
```

```
    auto lst3 = new List;                    // Ошибка!
       // Идентификатор List не определен!
}
```

Включение с переименованием не делает видимыми переименованные пакеты (то есть util, container, ..., list), так что попытка использовать исходное длинное имя в определении lst2 завершается неудачей при поиске первого же идентификатора util. Кроме того, включение с переименованием, без сомнения, обладает статической природой (см. раздел 11.1.5) в том смысле, что не использует механизм автоматического поиска; вот почему не вычисляется выражение new List. Если вы действительно хотите не только переименовать идентификаторы, но еще и сделать их видимыми, очень удобно использовать конструкцию alias (см. раздел 7.4):

```
import util.container.finite.linear.list;     // Нестатическое включение
alias util.container.finite.linear.list list; // Для удобства

void main() {
   auto lst1 = new list.List;                                  // Все в порядке
   auto lst2 = new util.container.finite.linear.list.List; // Все в порядке
   auto lst3 = new List;                                       // Все в порядке
}
```

Переименование также может использоваться в связке с избирательными включениями (см. раздел 11.1.6). Продемонстрируем это на примере:

```
import std.stdio : say = writeln;

void main() {
   say("Здравствуй, мир!");              // Все в порядке, вызвать writeln
   std.stdio.say("Здравствуй, мир");     // Ошибка!
   writeln("Здравствуй, мир!");          // Ошибка!
   std.stdio.writeln("Здравствуй, мир!"); // Ошибка!
}
```

Как и ожидалось, применив избирательное включение, которое одновременно еще и переименовывает идентификатор, вы делаете видимым лишь включаемый идентификатор и ничего больше.

Наконец, можно переименовать *и* модуль, *и* включаемый идентификатор (включаемые идентификаторы):

```
import io = std.stdio : say = writeln, CFile = File;
```

Возможные взаимодействия между двумя переименованными включенными идентификаторами могли бы вызвать некоторые противоречия. Язык D решил этот вопрос, просто сделав предыдущее объявление тождественным следующим:

```
import io = std.stdio : writeln, File;
import std.stdio : say = writeln, CFile = File;
```

Дважды переименовывающее объявление import эквивалентно двум другим объявлениям. Первое из этих объявлений переименовывает только

модуль, а второе – только включаемый идентификатор. Таким образом, новая семантика определяется в терминах более простых, уже известных видов инструкции `import`. Предыдущее определение вводит идентификаторы `io.writeln`, `io.File`, `say` и `CFile`.

11.1.8. Объявление модуля

Как говорилось в разделе 11.1, по той простой причине, что `import` принимает лишь идентификаторы, пакеты и модули на D, которые предполагается хоть когда-либо включать в другие модули с помощью этой конструкции, должны иметь имена, являющиеся допустимыми идентификаторами языка D.

В отдельных ситуациях требуется, чтобы модуль замаскировался именем, отличным от имени файла, где расположен код модуля, и притворился бы, что путь до пакета, которому принадлежит модуль, отличается от пути до каталога, где на самом деле располагается упомянутый файл. Очевидная ситуация, когда это может понадобиться: имя модуля не является допустимым идентификатором D.

Предположим, вы пишете программу, которая придерживается более широкого соглашения по именованию, предписывающего использовать дефисы в имени файла, например `gnome-cool-app.d`. Тогда компилятор D откажется компилировать ее, даже если сама программа будет полностью корректной. И все потому, что во время компиляции D должен генерировать информацию о каждом модуле, каждый модуль должен обладать допустимым именем, а `gnome-cool-app` не является таковым. Простой способ обойти это правило – хранить исходный код под именем `gnome-cool-app`, а на этапе сборки переименовывать его, например в `gnome_cool_app.d`. Этот трюк, конечно, сработает, но есть способ проще и лучше: достаточно вставить в начало файла объявление модуля, которое выглядит так:

```
module gnome_cool_app;
```

Если такое объявление присутствует в `gnome-cool-app.d` (но обязательно в качестве первого объявления в файле), то компилятор будет доволен, поскольку он генерирует всю информацию о модуле, используя имя `gnome_cool_app`. В таком случае истинное имя вообще никак не проверяется; в объявлении модуля имя может быть хоть таким:

```
module path.to.nonexistent.location.app;
```

Тогда компилятор сгенерирует всю информацию о модуле, как будто он называется `app.d` и расположен в каталоге `path/to/nonexistent/location`. Компилятору все равно, потому что он не обращается по этому адресу: поиск файлов ассоциируется исключительно с `import`, а здесь, при непосредственной компиляции `gnome-cool-app.d`, никаких включений нет.

11.1.9. Резюме модулей

Язык D поощряет модель разработки, которая не требует отделения объявлений от сущностей, определяемых программой (в C и C++ эти понятия фигурируют как «заголовки» и «исходные коды»). Вы просто располагаете код в модуле и включаете этот модуль в другие с помощью конструкции `import`. Тем не менее иногда хочется принять другую модель разработки, предписывающую более жесткое разделение между сигнатурами, которые модуль должен реализовать, и кодом, который стоит за этими сигнатурами. В этом случае потребуется работать с так называемыми *резюме модулей* (*module summaries*), построенными на основе исходного кода. Резюме модуля – это минимум того, что необходимо знать модулю о другом модуле, чтобы использовать его.

Резюме модуля – это фактически модуль без комментариев и реализаций функций. Реализации функций, использующих параметры времени компиляции, тем не менее в резюме модуля остаются. Ведь функции с параметрами времени компиляции должны быть доступны во время компиляции, так как могут быть вызваны непредвиденным образом в модуле-клиенте.

Резюме модуля состоит из корректного кода на D. Например:

```
/**
Это документирующий комментарий для этого модуля
*/
module acme.doitall;
/**
Это документирующий комментарий для класса A
*/
class A {
   void fun() {     }
   final void gun() {   . }
}

class B(T) {
   void hun() {     }
}

void foo() {
    ..
}

void bar(int n)(float x) {
   ...
}
```

При составлении резюме модуля `doitall` этот модуль копируется, но исключаются все комментарии, а тела всех функций заменяются на ; (исключение составляют функции с параметрами времени компиляции – такие функции остаются нетронутыми):

```
module acme.doitall;

class A {
   void fun();
   final void gun();
}

class B(T) {
   void hun() {     }
}

void foo();

void bar(int n)(float x) {

}
```

Резюме содержит информацию, необходимую другому модулю, чтобы использовать `acme.doitall`. В большинстве случаев резюме модулей автоматически вычисляются внутри работающего компилятора. Но компилятор может сгенерировать резюме по исходному коду и по вашему запросу (в случае эталонной реализации компилятора `dmd` для этого предназначен флаг `-H`). Сгенерированные резюме полезны, когда вы, к примеру, хотите распространить библиотеку в виде заголовков плюс скомпилированная библиотека.

Заметим, что исключение тел функций все же не гарантировано. Компилятор волен оставлять тела очень коротких функций в целях инлайнинга. Например, если функция `acme.doitall.foo` обладает пустым телом или просто вызывает другую функцию, ее тело может присутствовать в сгенерированном интерфейсном файле.

Подход к разработке, хорошо знакомый программистам на C и C++, заключается в сопровождении заголовочных файлов (то есть резюме) и файлов с реализациями вручную и по отдельности. Если вы изберете этот способ, работать придется несколько больше, но зато вы сможете поупражняться в коллективном руководстве. Например, право изменять заголовочные файлы может быть закреплено за командой проектировщиков, контролирующей все детали интерфейсов, которые модули предоставляют друг другу. А команду программистов, реализующих эти интерфейсы, можно наделить правом изменять файлы реализации и правом на чтение (но не изменение) заголовочных файлов, используемых в качестве текущей документации, направляющей процесс реализации. Компилятор проверяет, соответствует ли реализация интерфейсу (ну, по крайней мере синтаксически).

С языком D у вас есть выбор – вы можете: 1) вообще обойтись без резюме модулей, 2) разрешить компилятору сгенерировать их за вас, 3) сопровождать модули и резюме модулей вручную. Все примеры в этой книге выбирают вариант 1) – не использовать резюме модулей, оставив все заботы компилятору. Чтобы опробовать две другие возможности, вам

сначала потребуется организовать модули так, чтобы их иерархия соответствовала изображенной на рис. 11.2.

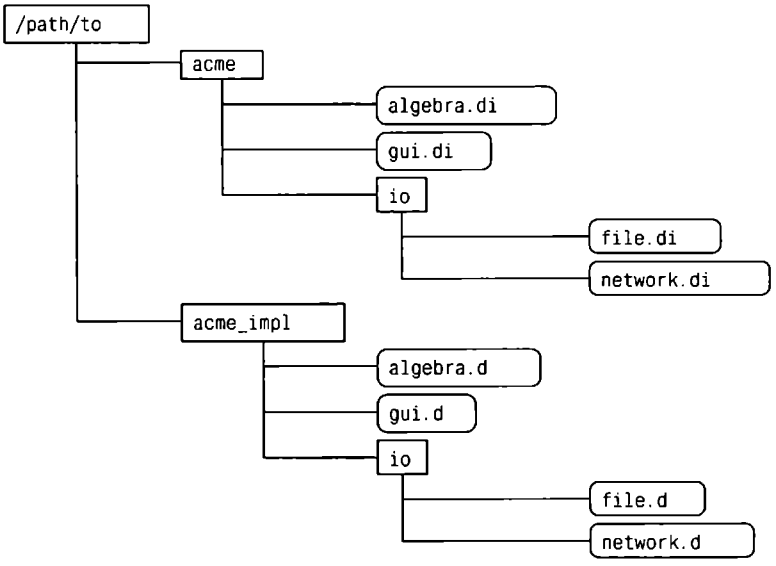

Рис. 11.2. Структура каталога для отделения резюме модулей («заголовков») от файлов реализации

Чтобы использовать пакет `acme`, потребуется добавить родительский каталог каталогов `acme` и `acme_impl` к базовым путям поиска модулей проекта (см. раздел 11.1.2), а затем включить модули из `acme` в клиентский код с помощью следующих объявлений:

```
// Из модуля client.d
import acme.algebra;
import acme.io.network;
```

Каталог `acme` включает только файлы резюме. Чтобы заставить файлы реализации взаимодействовать, необходимо, чтобы в качестве префикса в именах соответствующих модулей фигурировал пакет `acme`, а не `acme_impl`. Вот где приходят на помощь объявления модулей. Даже несмотря на то, что файл `algebra.d` находится в каталоге `acme_impl`, включив следующее объявление, модуль `algebra` может заявить, что входит в пакет `acme`:

```
// Из модуля acme_impl/algebra.d
module acme.algebra;
```

Соответственно модули в подпакете `io` будут использовать объявление:

```
// Из модуля acme_impl/io/file.d
module acme.io.file;
```

Эти строки позволят компилятору сгенерировать должные имена пакетов и модулей. Чтобы во время сборки программы компилятор нашел тела функций, просто передайте ему файлы реализации:

```
% dmd client.d /path/to/acme_impl/algebra.d
```

Директива `import` в `client.d` обнаружит интерфейсный файл `acme.di` в каталоге `/path/to/acme`. А компилятор найдет файл реализации точно там, где указано в командной строке, с корректными именами пакета и модуля.

Если коду из `client.d` потребуется использовать множество модулей из пакета `acme`, станет неудобно указывать все эти модули в командной строке компилятора. В таких случаях лучший вариант – упаковать весь код пакета `acme` в бинарную библиотеку и передавать `dmd` только ее. Синтаксис для сборки библиотеки зависит от реализации компилятора; если вы работаете с эталонной реализацией, вам потребуется сделать что-то типа этого:

```
% cd /path/to/acme_impl
% dmd -lib -ofacme algebra.d gui.d io/file.d io/network.d
```

Флаг `-lib` предписывает компилятору собрать библиотеку, а флаг `-of` (от output file – файл вывода) направляет вывод в файл `acme.lib` (Windows) или `acme.a` (UNIX-подобные системы). Чтобы клиентский код мог работать с такой библиотекой, нужно ввести что-то вроде:

```
% dmd client.d acme.lib
```

Если библиотека `acme` широко используется, ее можно сделать одной из библиотек, которые проект использует по умолчанию. Но тут уже многое зависит от реализации компилятора и от операционной системы, так что для успеха операции придется прочесть это жуткое руководство.

11.2. Безопасность

Понятие безопасности языков программирования всегда было противоречивым, но за последние годы его определение удивительно кристаллизовалось.

Интуитивно понятно, что безопасный язык тот, который «защищает свои собственные абстракции» [46, гл. 1]. В качестве примера таких абстракций D приведем класс:

```
class A { int x; }
```

и массив:

```
float[] array;
```

По правилам языка D (тоже «абстракция», предоставляемая языком) изменение внутреннего элемента `x` любого объекта типа `A` не должно изменять какой-либо элемент массива `array`, и наоборот, изменение `array[n]`

для некоторого n не должно изменять элемент x некоторого объекта типа A. Как ни благоразумно запрещать такие бессмысленные операции, в D есть способы заставить их обе выполниться – формируя указатели с помощью cast или задействуя union.

```
void main() {
   float[] array = new float[1024];
   auto obj = cast(A) array.ptr;

}
```

Изменение одного из элементов массива array (какого именно, зависит от реализации компилятора, но обычно второго или третьего) изменяет obj.x.

11.2.1. Определенное и неопределенное поведение

Кроме только что приведенного примера с сомнительным приведением указателя на float к ссылке на класс есть и другие ошибки времени исполнения, свидетельствующие о том, что язык нарушил определенные обещания. Хорошими примерами могут послужить разыменование указателя null, деление на ноль, а также извлечение вещественного квадратного корня из отрицательного числа. Никакая корректная программа не должна когда-либо выполнять такие операции, и тот факт, что они все же могут иметь место в программе, типы которой проверяются, можно рассматривать как несостоятельность системы типов.

Проблема подобного критерия корректности, который «хорошо было бы принять»: список ошибок бесконечно пополняется. D сводит свое понятие безопасности к очень точному и полезному определению: безопасная программа на D характеризуется только определенным поведением. Различия между определенным и неопределенным поведением:

- *определенное поведение:* выполнение фрагмента программы в заданном состоянии завершается одним из заранее определенных исходов; один из возможных исходов – резкое прекращение выполнения (именно это происходит при разыменовании указателя null и при делении на ноль);
- *неопределенное поведение:* эффект от выполнения фрагмента программы в заданном состоянии не определен. Это означает, что может произойти все, что угодно в пределах физических возможностей. Хороший пример – только что упомянутый случай с cast: программа с такой «раковой клеткой» некоторое время может продолжать работу, но наступит момент, когда какая-нибудь запись в array с последующим случайным обращением к obj приведет к тому, что исполнение выйдет из-под контроля.

(Неопределенное поведение перекликается с понятием недиагностированных ошибок, введенным Карделли [15]. Он выделяет две большие категории ошибок времени исполнения: диагностированные и недиаг-

ностированные ошибки. Диагностированные ошибки вызывают немедленный останов исполнения, а недиагностированные – выполнение произвольных команд. В программе с определенным поведением никогда не возникнет недиагностированная ошибка.)

У противопоставления определенного поведения неопределенному есть пара интересных нюансов. Рассмотрим, к примеру, язык, определяющий операцию деления на ноль с аргументами типа `int`, так что она должна всегда порождать значение `int.max`. Такое условие переводит деление на ноль в разряд определенного поведения – хотя данное определение этого действия и нельзя назвать полезным. Примерно в том же ключе `std.math` в действительности определяет, что операция `sqrt(-1)` должна возвращать `double.nan`. Это также определенное поведение, поскольку `double.nan` – вполне определенное значение, которое является частью спецификации языка, а также функции `sqrt`. Даже деление на ноль – не ошибка для типов с плавающей запятой: этой операции заботливо предписывается возвращать или плюс бесконечность, или минус бесконечность, или NaN («нечисло») (см. главу 2). Результаты выполнения программ всегда будут предсказуемыми, когда речь идет о функции `sqrt` или делении чисел с плавающей запятой.

Программа безопасна, если она не порождает неопределенное поведение.

11.3.2. Атрибуты @safe, @trusted и @system

Нехитрый способ гарантировать отсутствие недиагностированных ошибок – просто запретить все небезопасные конструкции D, например особые случаи применения выражения `cast`. Однако это означало бы невозможность реализовать на D многие системы. Иногда бывает очень нужно переступить границы абстракции, например, рассматривать область памяти, имеющей некоторый тип, как область памяти с другим типом. Именно так поступают менеджер памяти и сборщик мусора. В задачи языка D всегда входила способность выразить логику такого программного обеспечения на системном уровне.

С другой стороны, многие приложения нуждаются в небезопасном доступе к памяти лишь в сильно инкапсулированной форме. Язык может заявить о том, что он безопасен, даже если его сборщик мусора реализован на небезопасном языке. Ведь с точки зрения безопасного языка нет возможности использовать сборщик небезопасным образом. Сборщик сам по себе инкапсулирован внутри библиотеки поддержки времени исполнения, реализован на другом языке и воспринимается безопасным языком как волшебный примитив. Любой недостаток безопасности сборщика мусора был бы проблемой реализации языка, а не клиентского кода.

Как может большой проект обеспечить безопасность большинства своих модулей, в то же время обходя правила в некоторых избранных случаях? Подход D к безопасности – предоставить пользователю право самому

решать, чего он хочет: вы можете на уровне объявлений заявить, придерживается ли ваш код правил безопасности или ему нужна возможность переступить ее границы. Обычно информация о свойствах модуля указывается сразу же после объявления модуля, как здесь:

```
module my_widget;
@safe:
 ..
```

В этом месте определяются атрибуты `@safe`, `@trusted` и `@system`, которые позволяют модулю объявить о своем уровне безопасности. (Такой подход не нов; в языке Модула-3 применяется тот же подход, чтобы отличить небезопасные и безопасные модули.)

Код, размещенный после атрибута `@safe`, обязуется использовать инструкции лишь из безопасного подмножества D, что означает:

- никаких преобразований указателей в неуказатели (например, `int`), и наоборот;
- никаких преобразований между указателями, типы которых не имеют отношения друг к другу;
- проверка границ при любом обращении к массиву;
- никаких объединений, включающих указатели, классы и массивы, а также структуры, которые содержат перечисленные запрещенные типы в качестве внутренних элементов;
- никаких арифметических операций с указателями;
- запрет на получение адреса локальной переменной (на самом деле, требуется запрет утечки таких адресов, но отследить это гораздо сложнее);
- функции должны вызывать лишь функции, обладающие атрибутом `@safe` или `@trusted`;
- никаких ассемблерных вставок;
- никаких преобразований типа, лишающих данные статуса `const`, `immutable` или `shared`;
- никаких обращений к каким-либо сущностям с атрибутом `@system`.

Иногда эти правила могут оказаться излишне строгими; например, в стремлении избежать утечки указателей на локальные переменные можно исключить из рядов безопасных программ очевидно корректные программы. Тем не менее безопасное подмножество D (по прозвищу SafeD) все же довольно мощное – целые приложения могут быть полностью написаны на SafeD.

Объявление или группа объявлений могут заявить, что им, напротив, требуется низкоуровневый доступ. Такие объявления должны содержать атрибут `@system`:

```
@system:
void * allocate(size_t size);
```

```
void deallocate(void* p);
```

Атрибут @system действенно отключает все проверки, позволяя использовать необузданную мощь языка – на счастье или на беду.

Наконец, подход библиотек нередко состоит в том, что они предлагают клиентам безопасные абстракции, подспудно используя небезопасные средства. Такой подход применяют многие компоненты стандартной библиотеки D. В таких объявлениях можно указывать атрибут @trusted.

Модулям без какого-либо атрибута доступен уровень безопасности, назначаемый по умолчанию. Выбор уровня по умолчанию можно настроить с помощью конфигурационных файлов компилятора и флагов командной строки; точная настройка зависит от реализации компилятора. Эталонная реализация компилятора dmd предлагает атрибут по умолчанию @system; задать атрибут по умолчанию @safe можно с помощью флага командной строки -safe.

В момент написания этой книги SafeD находится в состоянии α-версии, так что порой небезопасные программы проходят компиляцию, а безопасные – нет, но мы активно работаем над решением этой проблемы.

11.3. Конструкторы и деструкторы модулей

Иногда модулям требуется выполнить какой-то инициализирующий код для вычисления некоторых статических данных. Сделать это можно, вставляя явные проверки («Были ли эти данные добавлены?») везде, где осуществляется доступ к соответствующим данным. Если такой подход неудобен/неэффективен, помогут конструкторы модулей.

Предположим, что вы пишете модуль, зависящий от операционной системы, и поведение этого модуля зависит от флага. Во время компиляции легко распознать основные платформы (например, «Я Mac» или «Я PC»), но определять версию Windows придется во время исполнения.

Чтобы немного упростить задачу, условимся, что наш код различает лишь ОС Windows Vista и более поздние или ранние версии относительно нее. Пример кода, определяющего вид операционной системы на этапе инициализации модуля:

```
private enum WinVersion { preVista, vista, postVista }
private WinVersion winVersion;

static this() {
   OSVERSIONINFOEX info;
   info.dwOSVersionInfoSize = OSVERSIONINFOEX.sizeof;
   GetVersionEx(&info) || assert(false);
   if (info.dwMajorVersion < 6) {
      winVersion = WinVersion.preVista;
```

```
    } else if (info.dwMajorVersion == 6 && info.dwMinorVersion == 0) {
       winVersion = WinVersion.vista;
    } else {
       winVersion = WinVersion.postVista;
    }
}
```

Этот геройский подвиг совершает конструктор модуля `static this()`. Такие *конструкторы модулей* всегда выполняются до `main`. Любой заданный модуль может содержать любое количество конструкторов.

В свою очередь, синтаксис деструкторов модулей предсказуем:

```
// На уровне модуля
static ~this() {

}
```

Статические деструкторы выполняются после того, как выполнение `main` завершится каким угодно образом, будь то нормальный возврат или порождение исключения. Модули могут определять любое количество деструкторов модуля и свободно чередовать конструкторы и деструкторы модуля.

11.3.1. Порядок выполнения в рамках модуля

Порядок выполнения конструкторов модуля в рамках заданного модуля всегда соответствует последовательности расположения этих конструкторов в модуле, то есть сверху вниз (лексический порядок). Порядок выполнения деструкторов модуля – снизу вверх (обратный лексический порядок).

Если один из конструкторов модуля не сможет выполниться и породит исключение, то не будет выполнена и функция `main`. Выполняются лишь статические деструкторы, лексически расположенные *выше* отказавшего конструктора модуля. Если не сможет выполниться и породит исключение какой-либо деструктор модуля, остальные деструкторы выполнены не будут, а приложение прекратит свое выполнение, выведя сообщение об ошибке в стандартный поток.

11.3.2. Порядок выполнения при участии нескольких модулей

Если модулей несколько, определить порядок вызовов сложнее. Эти правила идентичны определенным для статических конструкторов классов (см. раздел 6.3.6) и исходят из того, что модули, включаемые другими модулями, должны инициализироваться первыми, а очищаться – последними. Вот правила, определяющие порядок выполнения статических конструкторов модулей `модуль1` и `модуль2`:

- конструкторы или деструкторы модулей определяются только в одном из модулей модуль1 и модуль2, тогда не нужно заботиться об упорядочивании;
- модуль1 не включает модуль модуль2, а модуль2 не включает модуль1: упорядочивание не регламентируется – любой порядок сработает, поскольку модули не зависят друг от друга;
- модуль1 включает модуль2: конструкторы модуля2 выполняются до конструкторов модуля1, а деструкторы модуля2 – после деструкторов модуля1;
- модуль2 включает модуль1: конструкторы модуля1 выполняются до конструкторов модуля2, а деструкторы модуля1 – после деструкторов модуля2;
- модуль1 включает модуль2, а модуль2 вкючает модуль1: диагностируется ошибка «циклическая зависимость» и выполнение прерывается на этапе загрузки программы.

Проверка на циклическую зависимость модулей в настоящий момент делается во время исполнения. Такие циклы можно отследить и во время компиляции или сборки, но это мало что дает: проблема проявляется в том, что программа отказывается загружаться, и можно предположить, что перед публикацией программа запускается хотя бы один раз. Тем не менее чем раньше обнаружена проблема, тем лучше, так что язык оставляет реализации возможность выявить это некорректное состояние и сообщить о нем.

11.4. Документирующие комментарии

Писать документацию скучно, а для программиста нет ничего страшнее скуки. В результате документация обычно содержит скупые, неполные и устаревшие сведения.

Автоматизированные построители документации стараются вывести максимум информации из чистого кода, отразив заслуживающие внимания отношения между сущностями. Тем не менее современным автоматизированным построителям нелегко задокументировать высокоуровневые намерения по реализации. Современные языки помогают им в этом, предписывая использовать так называемые *документирующие комментарии* – особые комментарии, описывающие, например, определенную пользователем сущность. Языковой процессор (или сам компилятор, или отдельная программа) просматривает комментарии вместе с кодом и генерирует документацию в одном из популярных форматов (таком как XML, HTML или PDF).

D определяет для документирующих комментариев спецификацию, описывающую формат комментариев и процесс их преобразования в целевой формат. Сам процесс не зависит от целевого формата; транслятор,

управляемый простым и гибким шаблоном (также определяемым пользователем), генерирует документацию фактически в любом заданном формате.

Всеобъемлющее изучение системы трансляции документирующих комментариев не входит в задачу этой книги. Замечу только, что вам не помешает уделить этому больше внимания; документация многих проектов на D, а также веб-сайт эталонной реализации компилятора и его стандартной библиотеки полностью сгенерированы на основе документирующих комментариев D.

11.5. Взаимодействие с C и C++

Модули на D могут напрямую взаимодействовать с функциями C и C++. Есть ограничение: к этим функциям не относятся обобщенные функции C++, поскольку для этого компилятор D должен был бы включать полноценный компилятор C++. Кроме того, схема расположения полей класса D не совместима с классами C++, использующими виртуальное наследование.

Чтобы вызвать функцию C или C++, просто укажите в объявлении функции язык и не забудьте связать ваш модуль с соответствующими библиотеками:

```
extern(C) int foo(char*);
extern(C++) double bar(double);
```

Эти объявления сигнализируют компилятору, что вызов генерируется с соответствующими схемой расположения в стеке, соглашением о вызовах и кодировкой имен (также называемой декорированием имен – name mangling), даже если сами функции D отличаются по всем или некоторым из этих пунктов.

Чтобы вызвать функцию на D из программы на C или C++, просто добавьте в реализацию одно из приведенных выше объявлений:

```
extern(C) int foo(char*) {
    .. // Реализация
}
extern(C++) double bar(double) {
       // Реализация
}
```

Компилятор опять организует необходимое декорирование имен и использует соглашение о вызовах, подходящее для языка-клиента. То есть эту функцию можно с одинаковым успехом вызывать из модулей как на D, так и на «иностранных языках».

11.5.1. Взаимодействие с классами C++[1]

Как уже говорилось, D не способен отобразить классы C++ в классы D. Это связано с различием реализаций механизма наследования в этих языках. Тем не менее интерфейсы D очень похожи на классы C++, поэтому D реализует следующий механизм взаимодействия с классами C++:

```
// Код на C++

class Foo {
public:
   virtual int method(int a, int b) {
      return a + b;
   }
};

Foo* newFoo() {
   return new Foo();
}

void deleteFoo(Foo* obj) {
    delete obj;
}

// Код на D

extern (C++) {
   interface Foo {
      int method(int, int);
   }

   Foo newFoo();
   void deleteFoo(Foo);
}

void main() {
   auto obj = newFoo;
   scope(exit) deleteFoo(obj);
   assert(obj.method(2, 3) == 5);
}
```

Следующий код создает класс, реализующий интерфейс C++, и использует объект этого интерфейса в вызове внешней функции C++, принимающей в качестве аргумента указатель на объект класса C++ Foo.

[1] Описание этой части языка не было включено в оригинал книги, но поскольку данная возможность присутствует в текущих реализациях языка, мы добавили ее описание в перевод. – *Прим. науч. ред.*

```
extern (C++) void call(Foo);
// В коде C++ эта функция должна быть определена как void call(Foo* f);

extern (C++) interface Foo {
   int bar(int, int);
}

class FooImpl : Foo {
   extern (C++) int bar(int a, int b) {
      //
   }
}

void main() {
   FooImpl f = new FooImpl();
   call(f);
}
```

11.6. Ключевое слово deprecated

Перед любым объявлением (типа, функции или данных) может располагаться ключевое слово deprecated. Оно действует как класс памяти, но нисколько не влияет собственно на генерацию кода. Вместо этого deprecated лишь информирует компилятор о том, что помеченная им сущность не предназначена для использования. Если такая сущность все же будет использована, компилятор выведет предупреждение или даже откажется компилировать, если он был запущен с соответствующим флагом (-w в случае dmd).

Ключевое слово deprecated служит для планомерной постепенной миграции от старых версий API к более новым версиям. Причисляя соответствующие объявления к устаревшим, можно настроить компилятор так, чтобы он или принимал, или отклонял объявления с префиксом deprecated. Подготовив очередное изменение, отключите компиляцию deprecated – ошибки точно укажут, где требуется ваше вмешательство, что позволит вам шаг за шагом обновить код.

11.7. Объявления версий

В идеальном мире, как только программа написана, ее можно запускать где угодно. А здесь, на Земле, то и дело что-то заставляет вносить в программу изменения – другая версия библиотеки, сборка для особых целей или зависимость от платформы. Чтобы помочь справиться с этим, D определяет объявление версии version, позволяющее компилировать код в зависимости от определенных условий.

Способ использования версии намеренно прост и прямолинеен. Вы или устанавливаете версию, или проверяете ее. Сама версия может быть или целочисленной константой, или идентификатором:

```
version = 20100501;
version = FinalRelease;
```

Чтобы проверить версию, напишите:

```
version(20100501) {
       // Объявления
}

version (PreFinalRelease) {
   .. // Объявления
} else version (FinalRelease) {
       // Другие объявления
} else {
       // Еще объявления
}
```

Если версия уже присвоена, «охраняемые» проверкой объявления компилируются, иначе они игнорируются. Конструкция `version` может включать блок `else`, назначение которого очевидно.

Установить версию можно лишь до того, как она будет прочитана. Попытки установить версию после того, как она была задействована в проверке, вызывают ошибку времени компиляции:

```
version (ProEdition) {
    .. // Объявления
}
version = ProEdition; // Ошибка!
```

Реакция такова, поскольку присваивания версий не предназначены для того, чтобы версии изменять: версия должна быть одной и той же независимо от того, на какой фрагмент программы вы смотрите.

Указывать версию можно не только в файлах с исходным кодом, но и в командной строке компилятора (например, `-version=123` или `-version=xyz` в случае эталонной реализации компилятора `dmd`). Попытка установить версию как в командной строке, так и в файле с исходным кодом также приведет к ошибке.

Простота семантики `version` не случайна. Было бы легко сделать конструкцию `version` более мощной во многих отношениях, но очень скоро она начала бы работать наперекор своему предназначению. Например, управление версиями C с помощью связки директив `#if/#elif/#else`, безусловно, позволяет реализовать больше тактик в определении версий – именно поэтому управление версиями в проекте на C обычно содержит змеиный клубок условий, направляющих компиляцию. Конструкция `version` языка D намеренно ограничена, чтобы с ее помощью можно было реализовать лишь простое, единообразное управление версиями.

Компиляторы, как водится, имеют множество предопределенных версий, таких как платформа (например, `Win32`, `Posix` или `Mac`), порядок байтов (`LittleEndian`, `BigEndian`) и так далее. Если включено тестирование

модулей, автоматически задается проверка version(unittest). Особыми идентификаторами времени исполнения __FILE__ и __LINE__ обозначаются соответственно имя текущего файла и строка в этом файле. Полный список определений version приведен в документации вашего компилятора.

11.8. Отладочные объявления

Отладочное объявление – это лишь особая версия с идентичным синтаксисом присваивания и проверки. Конструкция debug была определена специально для того, чтобы стандартизировать порядок объявления отладочных режимов и средств.

Типичный случай использования конструкции debug:

```
module mymodule;

void fun() {
   int x;
   ..
   debug(mymodule) writeln("x="  x);
   ..
}
```

Чтобы отладить модуль mymodule, укажите в командной строке при компиляции этого модуля флаг -debug=mymodule, и выражение debug(mymodule) вернет true, что позволит скомпилировать код, «охраняемый» соответствующей конструкцией debug. Если использовать debug(5), то «охраняемый» этой конструкцией код будет включен при уровне отладки >= 5. Уровень отладки устанавливается либо присваиванием debug целочисленной константы, либо флагом компиляции. Допустимо также использовать конструкцию debug без аргументов. Код, следующий за такой конструкцией, будет добавлен, если компиляция запущена с флагом -debug. Как и в случае version, нельзя присваивать отладочной версии идентификатор после того, как он уже был проверен.

11.9. Стандартная библиотека D

Стандартная библиотека D, фигурирующая в коде под именем Phobos[1], органично развивалась вместе с языком. В результате она включает как API старого стиля, так и новейшие библиотечные артефакты, использующие более современные средства языка.

[1] Фобос (Phobos) – больший из двух спутников планеты Марс. «Марс» – изначальное название языка D (см. введение). Digital Mars (Цифровой Марс) – компания, разработавшая язык D и эталонную реализацию языка – компилятор dmd (от Digital Mars D). – *Прим. науч. ред.*

Библиотека состоит из двух основных пакетов – core и std. Первый содержит фундаментальные средства поддержки времени исполнения: реализации встроенных типов, сборщик мусора, код для начала и завершения работы, поддержка многопоточности, определения, необходимые для доступа к библиотеке времени исполнения языка C, и другие компоненты, связанные с перечисленными. Пакет std предоставляет функциональность более высокого уровня. Преимущество такого подхода в том, что другие библиотеки можно надстраивать поверх core, а с пакетом std они будут лишь сосуществовать, не требуя его присутствия.

Пакет std обладает плоской структурой: большинство модулей располагаются в корне пакета. Каждый модуль посвящен отдельной функциональной области. Информация о некоторых наиболее важных модулях библиотеки Phobos представлена в табл. 11.2.

Таблица 11.2. Обзор стандартных модулей

Модуль	Описание
std.algorithm	Этот модуль можно считать основой мощнейшей способности к обобщению, присущей языку. Вдохновлен стандартной библиотекой шаблонов C++ (Standard Template Library, STL). Содержит больше 70 важных алгоритмов, реализованных очень обобщенно. Большинство алгоритмов применяются к структурированным последовательностям идентичных элементов. В STL базовой абстракцией последовательности служит итератор, соответствующий примитив D – *диапазон*, для которого краткого обзора явно недостаточно; полное введение в диапазоны D доступно в Интернете [3]
std.array	Функции для удобства работы с массивами
std.bigint	Целое число переменной длины с сильно оптимизированной реализацией
std.bitmanip	Типы и часто используемые функции для низкоуровневых битовых операций
std.concurrency	Средства параллельных вычислений (см. главу 13)
std.container	Реализации разнообразных контейнеров
std.conv	Универсальный магазин, удовлетворяющий любые нужды по преобразованиям. Здесь определены многие полезные функции, такие как to и text
std.datetime	Полезные вещи, связанные с датой и временем
std.file	Файловые утилиты. Зачастую этот модуль манипулирует файлами целиком; например, в нем есть функция read, которая считывает весь файл, при этом std.file.read и понятия не имеет о том, что можно открывать файл и читать его маленькими порциями (об этом заботится модуль std.stdio, см. далее)
std.functional	Примитивы для определения и композиции функций

Модуль	Описание
std.getopt	Синтаксический анализ командной строки
std.json	Обработка данных в формате JSON
std.math	В высшей степени оптимизированные, часто используемые математические функции
std.numeric	Общие числовые алгоритмы
std.path	Утилиты для манипуляций с путями к файлам
std.random	Разнообразные генераторы случайных чисел
std.range	Определения и примитивы классификации, имеющие отношение к диапазонам
std.regex	Обработчик регулярных выражений
std.stdio	Стандартные библиотечные средства ввода/вывода, построенные на основе библиотеки stdio языка C. Входные и выходные файлы предоставляют интерфейсы в стиле диапазонов, благодаря чему многие алгоритмы, определенные в модуле std.algorithm, могут работать непосредственно с файлами
std.string	Функции, специфичные для строк. Строки тесно связаны с std.algorithm, так что модуль std.string, относительно небольшой по размеру, в основном лишь ссылается (определяя псевдонимы) на части std.algorithm, применимые к строкам
std.traits	Качества типов и интроспекция
std.typecons	Средства для определения новых типов, таких как Tuple
std.utf	Функции для манипулирования кодировками UTF
std.variant	Объявление типа Variant, который является контейнером для хранения значения любого типа. Variant – это высокоуровневый union

11.10. Встроенный ассемблер[1]

Строго говоря, большую часть задач можно решить, не обращаясь к столь низкоуровневому средству, как встроенный ассемблер, а те немногие задачи, которым без этого не обойтись, можно написать и скомпилировать отдельно, после чего скомпоновать с вашей программой на D обычным способом. Тем не менее встроенный в D ассемблер – очень мощное средство повышения эффективности кода, и упомянуть его необходимо. Конечно, в рамках одной главы невозможно всеобъемлюще описать язык ассемблера, да это и не нужно – ассемблеру для популярных платформ

[1] Описание этой части языка не было включено в оригинал книги, но поскольку эта возможность присутствует в текущих реализациях языка, мы добавили ее описание в перевод. – *Прим. науч. ред.*

посвящено множество книг[1]. Поэтому здесь мы приводим синтаксис и особенности применения встроенного ассемблера D, а описание используемых инструкций оставим специализированным изданиям.

К моменту написания данной книги компиляторы языка D существовали для платформ *x86* и *x86-64*, соответственно синтаксис встроенного ассемблера определен пока только для этих платформ.

11.10.1. Архитектура x86

Инструкции ассемблера можно встроить в код, разместив их внутри конструкции `asm`:

```
asm {
   naked;
   mov ECX, EAX;
   mov EAX, [ESP+size_t.sizeof*1];
   mov EBX, [ESP+size_t.sizeof*2];
L1:
   mov DH, [EBX + ECX - 1];
   mov [EAX + ECX - 1], DH;
   loop L1;
   ret;
}
```

Внутри конструкции `asm` допустимы следующие сущности:

- инструкция ассемблера:

  ‹инструкция› ‹арг$_1$›, ‹арг$_2$›, ..., ‹арг$_n$›;

- метка:

  ‹метка›:

- псевдоинструкция:

  ‹псевдоинструкция› ‹арг$_1$›, ‹арг$_2$›, ..., ‹арг$_n$›;

- комментарии.

Каждая инструкция пишется в нижнем регистре. После инструкции через запятую указываются аргументы. Инструкция обязательно завершается точкой с запятой. Несколько инструкций могут располагаться в одной строке. Метка объявляется перед соответствующей инструкцией как идентификатор метки с последующим двоеточием. Переход к метке может осуществляться с помощью оператора `goto` вне блока `asm`, а также с помощью инструкций семейства `jmp` и `call`. Аналогично внутри блока `asm` разрешается использовать метки, объявленные вне блоков `asm`. Комментарии в код на ассемблере вносятся так же, как и в остальном коде на D, другой синтаксис комментариев недопустим.

---

[1] Например, есть хороший учебник для вузов «Assembler» В. И. Юрова. – *Прим. науч. ред.*

Аргументом инструкции может быть идентификатор, объявленный вне блока asm, имя регистра, адрес (с применением обычных правил адресации данной платформы) или литерал соответствующего типа. Адреса можно записывать так (все эти адреса указывают на одно и то же значение):

```
mov EDX, 5[EAX][EBX];
mov EDX, [EAX+5][EBX];
mov EDX, [EAX+5+EBX];
```

Также разрешается использовать любые константы, известные на этапе компиляции, и идентификаторы, объявленные до блока asm:

```
int* p = arr.ptr;
asm
{
   mov EAX, p[EBP];              // Помещает в EAX значение p.
   mov EAX, p;                   // То же самое.
   mov EAX, [p + 2*int.sizeof]; // Помещает в EAX второй
                                 // элемент целочисленного массива.
}
```

Если размер операнда неочевиден, используется префикс *‹тип›* ptr:

```
add [EAX], 3;                    // Размер операнда 3 неочевиден.
add [EAX], int ptr 3;            // Теперь все ясно.
```

Префикс ptr можно использовать в сочетании с типами near, far, byte, short, int, word, dword, qword, float, double и real. Префикс far ptr не используется в плоской модели памяти D. По умолчанию компилятор использует byte ptr. Префикс seg возвращает номер сегмента адреса:

```
mov EAX seg p[EBP];
```

Этот префикс также не используется в плоской модели кода.

Также внутри блока asm доступны символы: $, указывающий на адрес следующей инструкции, и __LOCAL_SIZE, означающий количество байт в локальном кадре стека.

Для доступа к полю структуры, класса или объединения следует поместить адрес объекта в регистр и использовать полное имя поля в сочетании с offsetof:

```
struct Regs
{
   uint eax, ebx, ecx, edx;
}

void pushRegs(Regs* p)
{
   asm {
      push EAX;
      mov EAX, p;
```

```
            // Помещаем в p.ebx значение EBX
            mov [EAX+Regs.ebx.offsetof], EBX;

            // Помещаем в p.ecx значение ECX
            mov [EAX+Regs.ecx.offsetof], ECX;

            // Помещаем в p.edx значение EDX
            mov [EAX+Regs.edx.offsetof], EDX;

            pop EBX;

            // Помещаем в p.eax значение EAX
            mov [EAX+Regs.eax.offsetof], EBX;
        }
    }
```

Ассемблер *x*86 допускает обращение к следующим регистрам (имена регистров следует указывать заглавными буквами):

```
AL AH AX EAX                                BP EBP    ES CS SS DS GS FS
BL BH BX EBX                                SP ESP    CR0 CR2 CR3 CR4
CL CH CX ECX                                DI EDI    DR0 DR1 DR2 DR3 DR6 DR7
DL DH DX EDX                                SI ESI    TR3 TR4 TR5 TR6 TR7
ST
ST(0) ST(1) ST(2) ST(3) ST(4) ST(5) ST(6) ST(7)
MM0 MM1 MM2 MM3 MM4 MM5 MM6 MM7
XMM0 XMM1 XMM2 XMM3 XMM4 XMM5 XMM6 XMM7
```

Ассемблер D вводит следующие псевдоинструкции:

align *‹целочисленное_выражение›*;

целочисленное_выражение **должно вычисляться на этапе компиляции.** align **выравнивает следующую инструкцию по адресу, кратному** целочисленному_выражению**, вставляя перед этой инструкцией нужное количество инструкций** nop **(от Not OPeration), имеющих код** 0x90.

even;

Псевдоинструкция even **выравнивает следующую инструкцию по четному адресу (аналогична** align 2**). Выравнивание может сильно повлиять на производительность в циклах, где часто выполняется переход по выравниваемому адресу.**

naked;

Псевдоинструкция naked **указывает компилятору не генерировать пролог и эпилог функции. В прологе, как правило, создается новый кадр стека, а в эпилоге размещается код возвращения значения. Используя** naked**, программист должен сам позаботиться о получении**

нужных аргументов и возвращении результирующего значения в соответствии с применяемым функцией соглашением о вызовах.

Также ассемблер D разрешает вставлять в код непосредственные значения с помощью псевдоинструкций db, ds, di, dl, df, dd, de, которые соответствуют типам byte, short, int, long, float, double и extended и соответственно размещают значения этого типа (extended – тип с плавающей запятой длиной 10 байт, известный в D как real). Каждая такая псевдоинструкция может иметь насколько аргументов. Строковый литерал в качестве аргумента эквивалентен указанию n аргументов, где n – длина строки, а каждый аргумент соответствует одному знаку строки.

Следующий пример делает то же самое, что и первый пример в этом разделе:

```
asm {
   naked;
   db 0x89, 0xc1, 0x8b, 0x44, 0x24, 0x04, 0x8b;
   db 0x5c, 0x24, 0x08, 0x8a, 0x74, 0x0b, 0xff;
   db 0x88, 0x74, 0x08, 0xff, 0xe2, 0xf6, 0xc3; // Коротко и ясно.
}
```

Префиксы инструкций, такие как lock, rep, repe, repne, repnz и repz, указываются как отдельные псевдоинструкции:

```
asm
{
   rep;
   movsb;
}
```

Ассемблер D не поддерживает инструкцию pause. Вместо этого следует писать:

```
rep;
nop;
```

Для операций с плавающей запятой следует использовать формат с двумя аргументами.

```
fdiv ST(1);       // Неправильно
fmul ST;          // Неправильно
fdiv ST,ST(1);    // Правильно
fmul ST,ST(0);    // Правильно
```

11.10.2. Архитектура x86-64

Архитектура *x*86-64 является дальнейшим развитием архитектуры *x*86 и в большинстве случаев сохраняет обратную совместимость с ней. Рассмотрим отличия архитектуры *x*86-64 от *x*86.

Регистры общего назначения в *x*86-64 расширены до 64 бит. Их имена: RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, RIP и RFLAGS, причем RIP теперь доступен

из ассемблерного кода. Вдобавок добавились восемь 64-разрядных регистров общего назначения R8, R9, R10, R11, R12, R13, R14, R15. Для доступа к младшим 32 битам такого регистра к названию добавляется суффикс D, к младшим 16 – W, к младшим 8 – B. Так, R8D – младшие 4 байта регистра R8, а R15B – младший байт R15. Также добавились восемь XMM-регистров XMM8–XMM15.

Рассмотрим регистр RIP подробнее. Регистр RIP всегда содержит указатель на следующую инструкцию. Если в архитектуре *x*86, чтобы получить адрес следующей инструкции, приходилось писать код вида:

```
asm
{
   call $;          // Поместить в стек адрес следующей инструкции
                    // и передать на нее управление.
   pop EBX;         // Вытолкнуть адрес возврата в EBX.
   add EBX, 6;      // Скорректировать адрес на размер
                    // инструкций pop, add и mov.
   mov AL, [EBX];   // Теперь AL содержит код инструкции nop;
   nop;
}
```

то в *x*86-64 можно просто написать[1]:

```
asm
{
   mov AL, [RIP];   // Загружаем код следующей инструкции.
   nop;
}
```

К сожалению, выполнить переход по содержащемуся в RIP адресу с помощью jmp/jxx или call нельзя, равно как нельзя получить значение RIP, скопировав его в регистр общего назначения или стек. Впрочем, call $; как раз помещает в стек адрес следующей инструкции, что, по сути, идентично push RIP; (если бы такая инструкция была допустима). Подробную информацию можно найти в официальном руководстве по конкретному процессору.

11.10.3. Разделение на версии

По своей природе ассемблерный код является платформозависимым. Для *x*86 нужен один код, для *x*86-64 – другой, для SPARC – третий, а компилятор для виртуальной машины вообще может не иметь встроен-

[1] Ассемблер dmd2.052 не поддерживает доступ к регистру RIP. Возможно, данная функция появится позже. Ну а пока вместо mov AL, [RIP]; вы можете написать мантру db 0x8A, 0x05; di 0x00000000;, тем самым сообщив свое желание на языке процессора. Помните: если транслятор не понимает некоторые символы или инструкции, вы можете транслировать ассемблерный код в машинный сторонним транслятором и вставить в свой ассемблерный код числовое представление команды, воспользовавшись псевдоинструкциями семейства db. – *Прим. науч. ред.*

ного ассемблера. Хорошая практика – реализовать требуемую функциональность без использования ассемблера, добавив альтернативные реализации, оптимизированные для конкретных архитектур. Здесь пригодится механизм версий.

Компилятор dmd определяет версию D_InlineAsm_X86, если доступен ассемблер *x*86, и D_InlineAsm_X86_64 если доступен ассемблер *x*86-64.

Вот пример такого кода:

```
void optimizedFunction(void* arg) {
   version(D_InlineAsm_X86) {
      asm {
         naked;
         mov EBX, [EAX];
      }
   }
   else
   version(D_InlineAsm_X86_64) {
      asm {
         naked;
         mov RBX, [RAX];
      }
   }
   else {
      size_t s = *cast(size_t*)arg;
   }
}
```

11.10.4. Соглашения о вызовах

Все современные парадигмы программирования основаны на процедурной модели. Каким бы ни был ваш код – функциональным, объектно-ориентированным, агентно-ориентированным, многопоточным, распределенным, – он все равно будет вызывать процедуры. Разумеется, с повышением уровня абстракции, добавлением новых концепций процесс вызова процедур неизбежно усложняется.

Процедурный подход выгоден при организации взаимодействия фрагментов программы, написанных на разных языках. Во-первых, разные языки поддерживают разные парадигмы программирования, а во-вторых, даже одни и те же парадигмы могут быть реализованы по-разному. Между тем процедурный подход является тем самым фундаментом, на котором основано все остальное. Этот фундамент надежен, стандартизирован и проверен временем.

Вызов процедуры, как правило, состоит из следующих операций:

- передача аргументов;
- сохранение адреса возврата;
- переход по адресу процедуры;
- выполнение процедуры;

- передача возвращаемого значения;
- переход по сохраненному адресу возврата.

В высокоуровневом коде знать порядок выполнения этих операций необязательно, однако при написании кода на ассемблере их придется реализовывать самостоятельно.

То, как именно выполняются эти действия, определяется соглашениями о вызовах процедур. Их относительно немного, они хорошо стандартизированы. Разные языки используют разные соглашения о вызовах, но, как правило, допускают возможность использовать несколько соглашений. Соглашения о вызовах определяют, как передаются аргументы (через стек, через регистры, через общую память), порядок передачи аргументов, значение каких регистров следует сохранять, как передавать возвращаемое значение, кто возвращает указатель стека на исходную позицию (вызывающая или вызываемая процедура). В следующих разделах перечислены основные из этих соглашений.

11.10.4.1. Соглашения о вызовах архитектуры x86

Архитектура *x*86 за долгие годы своего существования породила множество соглашений о вызовах процедур. У каждого из них есть свои преимущества и недостатки. Все они требуют восстановления значений сегментных регистров.

cdecl

Данное соглашение принято в языке C, отсюда и его название (C Declaration). Большинство языков программирования допускают использование этого соглашения, и с его помощью наиболее часто организуется взаимодействие подпрограмм, написанных на разных языках. В языке D оно объявляется как функция с атрибутом `extern(C)`. Аргументы передаются через стек в обратном порядке, то есть начиная с последнего. Последним в стек помещается адрес возврата. Значение возвращается в регистре `EAX`, если по размеру оно меньше 4 байт, и на вершине стека, если его размер превышает 4 байта. В этом случае значение в `EAX` указывает на него. Если вы используете псевдоинструкцию `naked`, вам придется обрабатывать переданные аргументы вручную.

```
extern(C) int increment(int a) {
   asm {
      naked;
      mov EAX, [ESP+4]; // Помещаем в EAX значение a, смещенное на размер
                        // указателя (адреса возврата) от вершины стека.
      inc EAX;          // Инкрементируем EAX
      ret;              // Передаем управление вызывающей подпрограмме.
                        // Возвращаемое значение находится в EAX
   }
}
```

Стек восстанавливает вызывающая подпрограмма.

pascal

Соглашение о вызовах языка Паскаль в D объявляется как функция с атрибутом `extern(Pascal)`. Аргументы передаются в прямом порядке, стек восстанавливает вызываемая процедура. Значение возвращается через передаваемый неявно первый аргумент.

stdcall

Соглашение операционной системы Windows, используемое в WinAPI. Объявление: `extern(Windows)`. Аналогично `cdecl`, но стек восстанавливает вызываемая подпрограмма.

fastcall

Наименее стандартизированное и наиболее производительное соглашение о вызовах. Имеет две разновидности – `Microsoft fastcall` и `Borland fastcall`. В первом случае первые два аргумента в прямом порядке передаются через регистры `ECX` и `EDX`. Остальные аргументы передаются через стек в обратном порядке. Во втором случае через регистры `EAX`, `EDX` и `ECX` передаются первые три аргумента в прямом порядке, остальные аргументы передаются через стек в обратном порядке. В обоих случаях, если размер аргумента больше размера регистра, он передается через стек. Компиляторы D на данный момент не поддерживают данное соглашение, однако при использовании динамических библиотек есть возможность получить указатель на такую функцию и вызвать ее с помощью встроенного ассемблера.

thiscall

Данное соглашение обеспечивает вызов методов класса в языке C++. Полностью аналогично `stdcall`. Указатель на объект, метод которого вызывается, передается через `ECX`.

Соглашение языка D

Функция D гарантирует сохранность регистров `EBX`, `ESI`, `EDI`, `EBP`.

Если данная функция имеет постоянное количество аргументов, переменное количество гомогенных аргументов или это шаблонная функция с переменным количеством аргументов, аргументы передаются в прямом порядке и стек очищает вызываемая процедура. (В противном случае аргументы передаются в обратном порядке, после чего передается аргумент `_arguments`. `_argptr` не передается, он вычисляется на базе `_arguments`. Стек в этом случае очищает вызывающая процедура.) После этого в стеке резервируется пространство под возвращаемое значение, если оно не может быть возвращено через регистр. Последним передается аргумент `this`, если вызываемая процедура – метод структуры или класса, или указатель на контекст, если вызываемая процедура – делегат. Последний аргумент передается через регистр `EAX`, если он умещается в регистр, не является трехбайтовой структурой и не относится

к типу с плавающей запятой. Аргументы ref и out передаются как указатель, lazy – как делегат.

Возвращаемое значение передается так:

- bool, byte, ubyte, short, ushort, int, uint, 1-, 2- и 4-байтовые структуры, указатели (в том числе на объекты и интерфейсы), ссылки – в EAX;
- long, ulong, 8-байтовые структуры – в EDX (старшая часть) и EAX (младшая часть);
- float, double, real, ifloat, idouble, ireal – в ST0;
- cfloat, cdouble, creal – в ST1 (действительная часть) и ST0 (мнимая часть);
- динамические массивы – в EDX (указатель) и EAX (длина массива);
- ассоциативные массивы – в EAX;
- делегаты – в EDX (указатель на функцию) и EAX (указатель на контекст).

В остальных случаях аргументы передаются через скрытый аргумент, размещенный на стеке. В EAX в этом случае помещается указатель на этот аргумент.

11.10.4.2. Соглашения о вызовах архитектуры x86-64

С переходом к архитектуре *x*86-64 количество соглашений о вызовах существенно сократилось. По сути, осталось только два соглашения о вызовах – Microsoft x64 calling convention для Windows и AMD64 ABI convention для Posix.

Microsoft x64 calling convention

Это соглашение о вызовах очень напоминает fastcall. Аргументы передаются в прямом порядке. Первые 4 целочисленных аргумента передаются в RCX, RDX, R8, R9. Аргументы размером 16 байт, массивы и строки передаются как указатель. Первые 4 аргумента с плавающей запятой передаются через XMM0, XMM1, XMM2, XMM3. При этом место под эти аргументы резервируется в стеке. Остальные аргументы передаются через стек. Стек очищает вызывающая процедура. Возвращаемое значение передается в RAX, если оно умещается в 8 байт и не является числом с плавающей запятой. Число с плавающей запятой возвращается в XMM0. Если возвращаемое значение больше 64 бит, память под него выделяет вызывающая процедура, передавая ее как скрытый аргумент. В этом случае в RAX возвращается указатель на этот аргумент.

AMD64 ABI convention

Данное соглашение о вызовах используется в Posix-совместимых операционных системах и напоминает предыдущее, однако использует больше регистров. Для передачи целых чисел и адресов используются регистры RDI, RSI, RDX, RCX, R8 и R9, для передачи чисел с плавающей запятой –

XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 и XMM7. Если требуется передать аргумент больше 64 бит, но не больше 256 бит, он передается по частям через регистры общего назначения. В отличие от Microsoft x64, для переданных в регистрах аргументов место в стеке не резервируется. Возвращаемое значение передается так же, как и в Microsoft x64.

11.10.5. Рациональность

Решив применить встроенный ассемблер для оптимизации программы, следует понимать цену повышения эффективности. Ассемблерный код трудно отлаживать, еще труднее сопровождать. Ассемблерный код обладает плохой переносимостью. Даже в пределах одной архитектуры наборы инструкций разных процессоров несколько различаются. Более новый процессор может предложить более эффективное решение стоящей перед вами задачи. А раз уж вы добиваетесь максимальной производительности, то, возможно, предпочтете скомпилировать несколько версий своей программы для различных целевых архитектур, например одну переносимую версию, использующую только инструкции из набора *i*386, другую – для процессоров AMD, третью – для Intel Core. Для обычного высокоуровневого кода достаточно просто указать соответствующий флаг компиляции[1], а вот в случае с ассемблером придется создавать несколько версий кода, делающих одно и то же, но разными способами.

```
version(AMD)
{
   version = i686;
   version = i386;
}
else version(iCore)
{
   version = i686;
   version = i386;
}
else version(i686)
{
   version = i386;
}
```

[1] Компилятор dmd2.057 пока трудно назвать промышленным компилятором, поэтому упомянутого механизма в нем пока нет, а вот компилятор языка C gcc предоставляет возможность указать целевую платформу. Это позволяет получить максимально эффективный машинный код для данной платформы, при этом в код на языке C вносить изменения не нужно. Читателям, нуждающимся в компиляторах D, способных генерировать более оптимизированный код, следует обратить внимание на проекты GDC (GNU D compiler) и LDC (LLVM D compiler) компиляторов D, построенных на базе генераторов кода GCC и LLVM. – *Прим. науч. ред.*

```
void fastProcess()
{
   version(AMD) {
      //
    } else version(iCore) {
      //
   } else version(i686) {
      //
   } else version(i386)
 {
      //
   } else {
      //
   }
}
```

И все это ради того, чтобы выжать из функции `fastProcess` максимум производительности! Тут-то и надо задаться вопросом: а в самом ли деле эта функция является краеугольным камнем вашей программы? Может быть, ваша программа недостаточно производительна из-за ошибки на этапе проектирования, и выбор другого решения позволит сэкономить секунды процессорного времени – против долей миллисекунд, сэкономленных на оптимизации `fastProcess`? А может, время и, как следствие, деньги, которых требует написание ассемблерного кода, лучше направить на повышение производительности целевой машины? В любом случае задействовать встроенный ассемблер для повышения производительности нужно в последнюю очередь, когда остальные средства уже испробованы.

12

Перегрузка операторов

Мы, программисты, не очень любим слишком уж отделять встроенные типы от пользовательских. Магические свойства встроенных типов мешают открытости и расширяемости любого языка, поскольку при этом пользовательские типы обречены оставаться второсортными. Тем не менее у проектировщиков языка есть все законные основания относиться к встроенным типам с бóльшим почтением. Одно из таких оснований: более настраиваемый язык сложнее выучить, а также сложнее выполнять его синтаксический анализ как человеку, так и машине. Каждый язык по-своему определяет приемлемое соотношение между встроенным и настраиваемым, что для некоторых языков означает впадение в одну из этих двух крайностей.

Язык D подходит к этому вопросу прагматично: он не умаляет важность настраиваемости, но при этом осознает практичность встроенных типов – D использует преимущества встроенных типов ровно тремя путями:

1. *Синтаксис названий типов.* Массивы и ассоциативные массивы используются повсеместно, и, согласитесь, синтаксис `int[]` и `int[string]` гораздо нагляднее, чем `Array!int` и `AssotiativeArray!(string, int)`. В пользовательском коде нельзя определять новые формы записи названий типов, например `int[[]]`.

Литералы. Числовые и строковые литералы, как и литералы массивов и ассоциативных массивов, – «особые», и их набор нельзя расширить. «Сборные» объекты-структуры, такие как `Point(5, 3)`, – тоже литералы, но тип не может определить новый синтаксис литерала, например `(3, 5)pt`.

Семантика. Зная семантику определенных типов и их операций, компилятор оптимизирует код. Например, встретив выражение `"Hello" ~ " " ~ "world"`, компилятор не откладывает конкатенацию до

времени исполнения: он знает, что делает операция конкатенации строк, и склеивает строки уже во время компиляции. Аналогично компилятор упрощает и оптимизирует арифметические выражения, используя знание арифметики.

Некоторые языки добавляют к этому списку операторы. Они делают операторы особенными; чтобы выполнить какую-либо операцию применительно к пользовательским типам, приходится использовать стандартные средства языка, такие как вызов функций или макросов. Несмотря на то что это совершено законное решение, оно на самом деле создает проблемы при большом объеме кода, ориентированного на арифметические вычисления. Многие программы, ориентированные на вычисления, определяют собственные типы с алгебрами[1] (числа неограниченной точности, специализированные числа с плавающей запятой, кватернионы, октавы, матрицы всевозможных форм, тензоры, ... очевидно, что язык не может сделать встроенными их все). При использовании таких типов выразительность кода резко снижается. По сравнению с эквивалентным функциональным синтаксисом, операторы обычно требуют меньше места и круглых скобок, а получаемый с их участием код зачастую легок для восприятия. Рассмотрим для примера вычисление среднего гармонического трех ненулевых чисел x, y и z. Выражение на основе операторов очень близко к математическому определению:

```
m = 3 / (1/x + 1/y + 1/z);
```

В языке, требующем использовать вызовы функций вместо операторов, соответствующее выражение выглядит вовсе не так хорошо:

```
m = divide(3, add(add(divide(1, x), divide(1, y)), divide(1, z)));
```

Читать и изменять код с множеством арифметических функций гораздо сложнее, чем код с обычной записью операторов.

Язык D очень привлекателен для численного программирования. Он предоставляет надежную арифметику с плавающей запятой и превосходную библиотеку трансцендентных функций, которые иногда возвращают результат с большей точностью, чем «родные» системные библиотеки, и предлагает широкие возможности для моделирования. Мощное средство перегрузки операторов добавляет ему привлекательности. Перегрузка операторов позволяет вам определять собственные числовые типы (такие как числа с фиксированной запятой, десятичные числа для финансовых и бухгалтерских программ, неограниченные целые числа или действительные числа неограниченной точности), максимально близкие к встроенным числовым типам. Перегрузка операторов также позволяет определять типы с «числоподобными» алгебрами, такие как

[1] Автор использует понятия «тип» и «алгебра» не совсем точно. Тип определяет множество значений и множество операций, производимых над ними. Алгебра – это набор операций над определенным множеством. То есть уточнение «с алгебрами» – избыточно. – *Прим. науч. ред.*

векторы и матрицы. Давайте посмотрим, как можно определять типы с помощью этого средства.

12.1. Перегрузка операторов в D

Подход D к перегрузке операторов прост: если хотя бы один участник выражения с оператором имеет пользовательский тип, компилятор *заменяет* это выражение на обычный вызов метода с регламентированным именем. Затем применяются обычные правила языка. Таким образом, перегруженные операторы – лишь синтаксический сахар для вызова методов, а значит, нет нужды вникать в причуды самостоятельного средства языка. Например, если `a` относится к некоторому определенному пользователем типу, выражение `a + 5` заменяется на `a.opBinary!"+"(5)`. К методу `opBinary` применяются обычные правила и проверки, и тип `a` должен определять этот метод, если желает обеспечить поддержку перегрузки операторов.

Замена (точнее, *снижение*, т. к. этот процесс преобразует конструкции более высокого уровня в низкоуровневый код) – очень эффективный инструмент, позволяющий реализовать новые средства на основе имеющихся, и D обычно его применяет. Мы уже видели снижение в действии применительно к конструкции `scope` (см. раздел 3.13). По сути, `scope` – лишь синтаксический сахар, которым засыпаны особым образом сцепленные конструкции `try`, но вам точно не придет в голову самостоятельно писать сниженный код, так как `scope` значительно поднимает уровень высказываний. Перегрузка операторов действует в том же духе, определяя все вызовы операторов через замену на вызовы функций, тем самым придавая мощь обычным определениям функций и используя их как средство достижения своей цели. Без лишних слов посмотрим, как компилятор осуществляет снижение операторов разных категорий.

12.2. Перегрузка унарных операторов

В случае унарных операторов + (плюс), - (отрицание), ~ (поразрядное отрицание), * (разыменование указателя), ++ (увеличение на единицу) и -- (уменьшение на единицу) компилятор заменяет выражение

```
‹оп› a
```

на

```
a.opUnary!"‹оп›"()
```

для всех значений пользовательских типов. В качестве замены выступает вызов метода `opUnary` с одним аргументом времени компиляции `"‹оп›"` и без каких-либо аргументов времени исполнения. Например `++a` перезаписывается как `a.opUnary!"++"()`.

Чтобы перегрузить один или несколько унарных операторов для типа `T`, определите метод `T.opUnary` так:

```
struct T {
   SomeType opUnary(string op)();
}
```

В таком виде, как он здесь определен, этот метод будет вызываться для всех унарных операторов. А если вы хотите для некоторых операторов определить отдельные методы, вам помогут ограничения сигнатуры (см. раздел 5.4). Рассмотрим определение типа `CheckedInt`, который служит оберткой базовых числовых типов и гарантирует, что значения, получаемые в результате применения операций к оборачиваемым типам, не выйдут за границы, установленные для этих типов. Тип `CheckedInt` должен быть параметризирован оборачиваемым типом (например, `CheckedInt!int`, `CheckedInt!long` и т. д.). Вот неполное определение `CheckedInt` с операторами префиксного увеличения и уменьшения на единицу:

```
struct CheckedInt(N) if (isIntegral!N) {
   private N value;
   ref CheckedInt opUnary(string op)() if (op == "++") {
      enforce(value != value.max);
      ++value;
      return this;
   }
   ref CheckedInt opUnary(string op)() if (op == "--") {
      enforce(value != value.min);
      --value;
      return this;
   }
   .
}
```

12.2.1. Объединение определений операторов с помощью выражения mixin

Есть очень мощная техника, позволяющая определить не один, а сразу группу операторов. Например, все унарные операторы `+`, `-` и `~` для типа `CheckedInt` делают одно и то же – всего лишь проталкивают соответствующую операцию по направлению к `value`, внутреннему элементу `CheckedInt`. Хоть эти операторы и неидентичны, они несомненно придерживаются одного и того же шаблона поведения. Можно просто определить специальный метод для каждого оператора, но это вылилось бы в неинтересное дублирование шаблонного кода. Лучший подход – использовать работающие со строками выражения `mixin` (см. раздел 2.3.4.2), позволяющие напрямую монтировать операции из имен операндов и идентификаторов операторов. Следующий код реализует все унарные операции, применимые к `CheckedInt`.[1]

[1] В данном коде отсутствует проверка перехода за границы для оператора отрицания. – *Прим. науч. ред.*

```
struct CheckedInt(N) if (isIntegral!N) {
   private N value;
   this(N value) {
      this.value = value;
   }
   CheckedInt opUnary(string op)()
         if (op == "+" || op == "-" || op == "~") {
      return CheckedInt(mixin(op ~ "value"));
   }

   ref CheckedInt opUnary(string op)() if (op == "++" || op == "--") {
      enum limit = op == "++" ? N.max : N.min;
      enforce(value != limit);
      mixin(op ~ "value;");
      return this;
   }

}
```

Это уже заметная экономия на длине кода, и она лишь возрастет, как только мы доберемся до бинарных операторов и выражений индексации. Главное действующее лицо такого подхода – выражение `mixin`, которое позволяет вам взять строку и попросить компилятор скомпилировать ее. Строка получается буквальным соединением операнда и оператора вручную. Способность трансформироваться в код, по счастливой случайности присущая строке `op`, фактически воплощает в жизнь эту идиому; на самом деле, весь этот механизм перегрузки проектировался с прицелом на `mixin`. Раньше D использовал отдельное имя для каждого оператора (`opAdd`, `opSub`, `opMul`, ...), что требовало механического запоминания соответствия имен операторам и написания группы функций с практически идентичными телами.

12.2.2. Постфиксный вариант операторов увеличения и уменьшения на единицу

Постфиксные операторы увеличения (`a++`) и уменьшения (`a--`) на единицу – необычные: они выглядят так же, как и их префиксные «коллеги», так что различать их по идентификатору не получится. Дополнительная проблема в том, что вызывающему коду, которому нужен результат применения оператора, также должно быть доступно и старое значение сущности, увеличенной на единицу. Наконец, постфиксные и префиксные варианты операторов увеличения и уменьшения на единицу должны согласовываться друг с другом.

Постфиксное увеличение и уменьшение на единицу можно целиком сгенерировать из префиксного увеличения и уменьшения на единицу соответственно – нужно лишь немного шаблонного кода. Но вместо того чтобы заставлять вас писать этот шаблонный код, D делает это сам.

Замена a++ выполняется так (постфиксное уменьшение на единицу обрабатывается аналогично):

- если результат a++ не используется, осуществляется замена на ++a, что затем перезаписывается на a.opUnary!"++"();
- если результат a++ используется (например, arr[a++]), заменой послужит выражение (тяжкий вздох) ((ref x) {auto t=x; ++x; return t;})(a).

В первом случае попросту обыгрывается тот факт, что постфиксный оператор увеличения на единицу без применения результата делает то же самое, что и префиксный вариант соответствующего оператора. Во втором случае определяется лямбда-функция (см. раздел 5.6), выполняющая нужный шаблонный код: она создает новую копию входных данных, прибавляет единицу к входным данным и возвращает созданную ранее копию. Лямбда-функция применяется непосредственно к увеличиваемому значению.

12.2.3. Перегрузка оператора cast

Явное приведение типов осуществляется с помощью унарного оператора, применение которого выглядит как cast(T) a. Он немного отличается от всех остальных операторов тем, что использует тип в качестве параметра, а потому для него выделена особая форма снижения. Для любого значения пользовательского типа и некоторого другого типа T приведение

```
cast(T) значение
```

переписывается как

```
значение.opCast!T()
```

Реализация метода opCast, разумеется, должна возвращать значение типа T – деталь, на которой настаивает компилятор. Несмотря на то что перегрузка функций по значению аргумента не обеспечивается на уровне средства языка, множественные определения opCast можно реализовать с помощью шаблонов с ограничениями сигнатуры. Например, методы приведения к типам string и int для некоторого типа T можно определить так:

```
struct T {
   string opCast(T)() if (is(T == string)) {
        .
   }
   int opCast(T)() if (is(T == int)) {
      ..
   }
}
```

Можно определить приведение к целому классу типов. «Надстроим» пример с CheckedInt, определив приведение ко всем встроенным числовым типам. Загвоздка в том, что некоторые из них могут обладать более

ограничивающим диапазоном значений, а нам бы хотелось гарантировать, что преобразование не будет сопровождаться никакими потерями информации. Дополнительная задача: хотелось бы избежать проверок там, где они не требуются (например, нет нужды проверять границы при преобразовании из CheckedInt!int в long). Поскольку информация о границах доступна во время компиляции, вставка проверок лишь там, где это необходимо, задается с помощью конструкции static if (см. раздел 3.4):

```
struct CheckedInt(N) if (isIntegral!N) {
   private N value;
   // Преобразования ко всевозможным целым типам
   N1 opCast(N1)() if (isIntegral!N1) {
      static if (N.min < N1.min) {
         enforce(N1.min <= value);
      }
      static if (N.max > N1.max) {
         enforce(N1.max >= value);
      }
      // Теперь можно без опаски делать "сырые" преобразования
      return cast(N1) value;
   }
   ...
}
```

12.2.4. Перегрузка тернарной условной операции и ветвления

Встретив значение пользовательского типа, компилятор заменяет код вида

```
a ? ‹выражение1› : ‹выражение2›
```

на

```
cast(bool) a ? ‹выражение1› : ‹выражение2›
```

Сходным образом компилятор переписывает проверку внутри конструкции if с

```
if (a) ‹инструкция› // С блоком else или без него
```

на

```
if (cast(bool) a) ‹инструкция›
```

Оператор отрицания ! также переписывается в виде отрицания выражения с cast.

Чтобы обеспечить выполнение таких проверок, определите метод приведения к типу bool, как это сделано здесь:

```
struct MyArray(T) {
   private T[] data;
   bool opCast(T)() if (is(T == bool)) {
```

```
        return !data.empty;
    }
    ..
}
```

12.3. Перегрузка бинарных операторов

В случае бинарных операторов + (сложение), - (вычитание), * (умножение), / (деление), % (получение остатка от деления), & (поразрядное И), | (поразрядное ИЛИ), << (сдвиг влево), >> (сдвиг вправо), ~ (конкатенация) и in (проверка на принадлежность множеству) выражение

```
a ‹оп› b
```

где по крайней мере один из операндов a и b имеет пользовательский тип, переписывается в виде

```
a.opBinary!"‹оп›"(b)
```

или

```
b.opBinaryRight!"‹оп›"(a)
```

Если разрешение имен и проверки перегрузки успешны лишь для одного из этих вызовов, он выбирается для замены. Если оба вызова допустимы, возникает ошибка в связи с двусмысленностью. Если же не подходит ни один из вызовов, очевидно, что перед нами ошибка «идентификатор не найден».

Продолжим наш пример с CheckedInt из раздела 12.2. Определим для этого типа все бинарные операторы:

```
struct CheckedInt(N) if (isIntegral!N) {
    private N value;
    // Сложение
    CheckedInt opBinary(string op)(CheckedInt rhs) if (op == "+") {
        auto result = value + rhs.value;
        enforce(rhs.value >= 0 ? result >= value : result < value);
        return CheckedInt(result);
    }
    // Вычитание
    CheckedInt opBinary(string op)(CheckedInt rhs) if (op == "-") {
        auto result = value - rhs.value;
        enforce(rhs.value >= 0 ? result <= value : result > value);
        return CheckedInt(result);
    }
    // Умножение
    CheckedInt opBinary(string op)(CheckedInt rhs) if (op == "*") {
        auto result = value * rhs.value;
        enforce(value && result / value == rhs.value ||
            rhs.value && result / rhs.value == value ||
            result == 0);
```

```
        return CheckedInt(result);
    }
    // Деление и остаток от деления
    CheckedInt opBinary(string op)(CheckedInt rhs)
          if (op == "/" || op == "%") {
        enforce(rhs.value != 0);
        return CheckedInt(mixin("value" ~ op ~ "rhs.value"));
    }
    // Сдвиг
    CheckedInt opBinary(string op)(CheckedInt rhs)
          if (op == "<<" || op == ">>" || op == ">>>") {
        enforce(rhs.value >= 0 && rhs.value <= N.sizeof * 8);
        return CheckedInt(mixin("value" ~ op ~ "rhs.value"));
    }
    // Поразрядные операции (без проверок, переполнение невозможно)
    CheckedInt opBinary(string op)(CheckedInt rhs)
          if (op == "&" || op == "|" || op == "^") {
        return CheckedInt(mixin("value" ~ op ~ "rhs.value"));
    }
    ..
}
```

(Многие из этих проверок можно осуществить дешевле – с помощью бита переполнения, имеющегося у процессоров Intel, который при выполнении арифметических операций или устанавливается, или сбрасывается. Но это аппаратно-зависимый способ.) Данный код определяет по одному отдельному оператору для каждой уникальной проверки. Если у двух и более операторов одинаковый код, они всегда объединяются в один метод. Это сделано в случае операторов / и % (поскольку оба они выполняют одну и ту же проверку), всех операторов сдвига и трех поразрядных операторов, не требующих проверок. Здесь снова применен подход, смысл которого – собрать операцию в виде строки, а потом с помощью `mixin` скомпилировать ее в выражение.

12.3.1. Перегрузка операторов в квадрате

Если перегрузка операторов означает разрешение типам определять собственную реализацию операторов, то перегрузка перегрузки операторов, то есть перегрузка операторов в квадрате, означает разрешение типам определять некоторое количество перегруженных версий перегруженных операторов.

Рассмотрим выражение `a * 5`, где операнд `a` имеет тип `CheckedInt!int`. Оно не скомпилируется, поскольку до сих пор тип `CheckedInt` определял метод `opBinary` с правым операндом типа `CheckedInt`. Так что для выполнения вычисления в клиентском коде нужно писать `a * CheckedInt!int(5)`, что довольно неприятно.

Верный способ решить эту проблему – определить еще одну или несколько дополнительных реализаций метода `opBinary` для типа `CheckedInt!N`, так чтобы на этот раз тип `N` ожидался справа от оператора. Может

показаться, что определение нового метода opBinary потребует изрядного объема монотонной работы, но на самом деле достаточно добавить всего одну строчку:

```
struct CheckedInt(N) if (isIntegral!N) {
       // То же, что и раньше
   // Операции с "сырыми" числами
   CheckedInt opBinary(string op)(N rhs) {
      return opBinary!op(CheckedInt(rhs));
   }
}
```

Красота этого подхода в простоте: оператор преобразуется в обычный идентификатор, который затем можно передать другой реализации оператора.

12.3.2. Коммутативность

Присутствие opBinaryRight требуется в тех случаях, когда тип, определяющий оператор, является правым операндом, например, как в выражении 5 * a. В этом случае тип операнда a имеет шанс «поймать» оператор, лишь определив метод opBinaryRight!"*"(int). Здесь есть некоторая избыточность – если, скажем, нужно организовать поддержку операций, для которых не важно, с какой стороны подставлен целочисленный операнд (например, все равно, 5 * a или a * 5), вам потребуется определить как opBinary!"*"(int), так и opBinaryRight!"*"(int), а это расточительство, т. к. умножение коммутативно. При этом, предоставив языку принимать решение о коммутативности, можно столкнуться с излишними ограничениями: свойство коммутативности зависит от алгебры; например, умножение матриц некоммутативно. Поэтому компилятор оставляет за пользователем право определить отдельные операторы для правого и левого операндов, отказываясь брать на себя какую-либо ответственность за коммутативность операторов.

Чтобы организовать поддержку a *‹оп›* b и b *‹оп›* a, когда один операнд легко преобразуется к типу другого операнда, достаточно добавить одну строку:

```
struct CheckedInt(N) if (isIntegral!N) {
       // То же, что и раньше
   // Реализовать правосторонние операторы
   CheckedInt opBinaryRight(string op)(N lhs) {
      return CheckedInt(lhs).opBinary!op(this);
   }
}
```

Все, что нужно, – получить соответствующее выражение с CheckedInt слева. А затем вступают в права уже определенные операторы.

Но иногда для преобразования требуется ряд дополнительных шагов, без которых можно было бы обойтись. Например, представьте выражение 5 * c, где c имеет тип Complex!double. Применив приведенное вы-

ше решение, мы бы протолкнули умножение в выражение `Complex!double(5) * c`, при вычислении которого пришлось бы преобразовать 5 в комплексное число с нулевой мнимой частью, а затем зачем-то умножать комплексные числа, когда достаточно было бы всего лишь двух умножений действительных чисел. Результат, конечно, будет верным, но для его получения пришлось бы гораздо больше попотеть. В таких случаях лучше всего разделить правосторонние операции на две группы – коммутативные и некоммутативные операции – и обрабатывать их по отдельности. Коммутативные операции можно обрабатывать просто с помощью перестановки аргументов. Некоммутативные операции можно реализовывать так, чтобы каждый случай обрабатывался отдельно – или каждый раз заново, или извлекая пользу из уже реализованных примитивов.

```
struct Complex(N) if (isFloatingPoint!N) {
   N re, im;
   // Реализовать коммутативные операторы
   Complex opBinaryRight(string op)(N lhs)
      if (op == "+" || op == "*")
   {
      // Предполагается, что левосторонний оператор уже реализован
      return opBinary!op(lhs);
   }
   // Реализовать некоммутативные операторы вручную
   Complex opBinaryRight(string op)(N lhs) if (op == "-") {
      return Complex(lhs - re, -im);
   }
   Complex opBinaryRight(string op)(N lhs) if (op == "/") {
      auto norm2 = re * re + im * im;
      enforce(norm2 != 0);
      auto t = lhs / norm2;
      return Complex(re * t, -im * t);
   }
}
```

Для других типов можно выбрать другой способ группировки некоторых групп операций, в таком случае могут пригодиться уже описанные техники наложения ограничений на `op`.

12.4. Перегрузка операторов сравнения

В случае операторов сравнения (равенство и упорядочивание) D следует той же схеме, с которой мы познакомились, обсуждая классы (см. разделы 6.8.3 и 6.8.4). Может показаться, что так сложилось исторически, но есть и веские причины обрабатывать сравнения не в общем методе `opBinary`, а иным способом. Во-первых, между операторами `==` и `!=` есть тесные взаимоотношения, как и у всей четверки `<`, `<=`, `>` и `>=`. Эти взаимоотношения подразумевают, что лучше использовать две отдельные функции со специфическими именами, чем код, определяющий каж-

дый оператор отдельно в зависимости от идентификаторов. Кроме того, многие типы, скорее всего, будут определять лишь равенство и упорядочивание, а не все возможные операторы. С учетом этого факта для определения сравнений язык предоставляет простое и компактное средство, не заставляя использовать мощный инструмент `opBinary`.

Замена `a == b`, где хотя бы один из операндов `a` и `b` имеет пользовательский тип, производится по следующему алгоритму:

- Если как `a`, так и `b` – экземпляры классов, заменой служит выражение `object.opEquals(a, b)`. Как говорилось в разделе 6.8.3, сравнения между классами подчиняются небольшому протоколу, реализованному в модуле `object` из ядра стандартной библиотеки.
- Иначе если при разрешении имен `a.opEquals(b)` и `b.opEquals(a)` выясняется, что это обращения к одной и той же функции, заменой служит выражение `a.opEquals(b)`. Такое может произойти, если `a` и `b` имеют один и тот же тип, с одинаковыми или разными квалификаторами.
- Иначе компилируется только одно из выражений `a.opEquals(b)` и `b.opEquals(a)`, которое и становится заменой.

Выражения с каким-либо из четырех операторов упорядочивающего сравнения `<`, `<=`, `>` и `>=` переписываются по следующему алгоритму:

- Если при разрешении имен `a.opCmp(b)` и `b.opCmp(a)` выясняется, что это обращения к одной и той же функции, заменой служит выражение `a.opCmp(b)` *‹оп›* `0`.
- Иначе компилируется только одно из выражений `a.opCmp(b)` и `b.opCmp(a)`. Если первое, то заменой служит выражение `a.opCmp(b)` *‹оп›* `0`. Иначе заменой служит выражение `0` *‹оп›* `b.opCmp(a)`.

Здесь также стоит упомянуть о разумном обосновании одновременного существования как `opEquals`, так и `opCmp`. На первый взгляд может показаться, что достаточно и одного метода `opCmp` (равенство было бы реализовано как `a.opCmp(b) == 0`). Но хотя большинство типов могут определить равенство, многим типам нелегко реализовать отношение неравенства. Например, матрицы и комплексные числа определяют равенство, однако канонического определения отношения порядка им недостает.

12.5. Перегрузка операторов присваивания

К операторам присваивания относится не только простое присваивание вида `a = b`, но и присваивания с выполнением «на ходу» бинарных операторов, например `a += b` или `a *= b`. В разделе 7.1.5.1 уже было показано, что выражение

```
a = b
```

переписывается как

```
a.opAssign(b)
```

При выполнении бинарных операторов «на месте» заменой

```
a ‹оп›= b
```

послужит

```
a.opOpAssign!"‹оп›"(b)
```

Замена позволяет типу операнда a реализовать операции «на месте» по описанным выше техникам. Рассмотрим пример реализации оператора += для типа CheckedInt:

```
struct CheckedInt(N) if (isIntegral!N) {
   private N value;
   ref CheckedInt opOpAssign(string op)(CheckedInt rhs)
         if (op == "+") {
      auto result = value + rhs.value;
      enforce(rhs.value >= 0 ? result >= value : result <= value);
      value = result;
      return this;
   }

}
```

В этом определении примечательны три детали. Во-первых, метод opAssign возвращает ссылку на текущий объект, благодаря чему поведение CheckedInt становится сравнимым с поведением встроенных типов. Во-вторых, истинное вычисление делается не «на месте», а напротив, «в сторонке». Собственно, состояние объекта изменяется лишь после удачного выполнения проверки. В противном случае, если при вычислении выражения с enforce будет порождено исключение, мы рискуем испортить текущий объект. В-третьих, тело оператора фактически дублирует тело метода opBinary!"+", рассмотренного выше. Воспользуемся последним наблюдением, чтобы задействовать имеющиеся реализации всех бинарных операторов в определении операторов присваивания, одновременно выполняющих и бинарные операции. Вот новое определение:

```
struct CheckedInt(N) if (isIntegral!N) {
      // То же, что и раньше
   // Определить все операторы присваивания
   ref CheckedInt opOpAssign(string op)(CheckedInt rhs) {
      value = opBinary!op(rhs).value;
      return this;
   }
}
```

Можно было бы поступить и по-другому: определять бинарные операторы через операторы присваивания, определяемые с нуля. К этому выбору можно прийти из соображений эффективности; для многих типов изменение объекта «на месте» требует меньше памяти и выполняется быстрее, чем создание нового объекта.

12.6. Перегрузка операторов индексации

Язык D позволяет определять полностью абстрактный массив – массив, который поддерживает все операции, обычно ожидаемые от массива, но никогда не предоставляет адреса своих элементов клиентскому коду. Перегрузка операторов индексации – необходимое условие реализации этого средства. Чтобы обеспечить должный доступ по индексу, компилятор различает чтение и запись элементов. В последнем случае элемент массива находится слева от оператора присваивания, простой ли это оператор = или выполняющийся «на месте» бинарный оператор, такой как +=.

Если никакого присваивания не выполняется, компилятор заменяет выражение

```
a[b1, b2, .., bk]
```

на

```
a.opIndex(b1, b2, .. bk)
```

для любого числа аргументов *k*. Сколько принимается аргументов, какими должны быть их типы и каков тип результата, решает реализация метода `opIndex`.

Если результат применения оператора индексации участвует в присваивании слева, при снижении выражение

```
a[b1, b2, bk] = c
```

преобразуется в

```
a.opIndexAssign(c, b1, b2, .. bk)
```

Если к результату выражения с индексом применятся оператор увеличения или уменьшения на единицу, выражение

```
‹оп› a[b1, b2, .., bk]
```

где в качестве *‹оп›* выступает или ++, --, или унарный -, +, ~, *, переписывается как

```
a.opIndexUnary!"‹оп›"(b1, b2, bk)
```

Постфиксные увеличение и уменьшение на единицу генерируются автоматически из соответствующих префиксных вариантов, как описано в разделе 12.2.2.

Наконец, если полученный по индексу элемент изменяется «на месте», при снижении выражение

```
a[b1, b2, .., bk] ‹оп›= c
```

преобразуется в

```
a.opIndexOpAssign!"‹оп›"(c, b1, b2, bk)
```

Эти замены позволяют типу операнда a полностью определить, каким образом выполняется доступ к элементам, получаемым по индексу, и как они обрабатываются. Для чего индексируемому типу брать на себя ответственность за операторы присваивания? Казалось бы, более удачное решение – просто предоставить методу opIndex возвращать ссылку на хранимый элемент, например:

```
struct MyArray(T) {
   ref T opIndex(uint i) {     }
}
```

Тогда какие бы операции присваивания и изменения-с-присваиванием ни поддерживал тип T, они будут выполняться правильно. Предположим, дан массив типа MyArray!int с именем a, тогда при вычислении выражения a[7] *= 2 сначала с помощью метода opIndex будет получено значение типа ref int, а затем эта ссылка будет использована для умножения «на месте» на 2. На самом деле, именно так и работают встроенные массивы.

К сожалению, это решение не без изъяна. Одна из проблем заключается в том, что немалое число коллекций, построенных по принципу массива, не пожелают предоставить доступ к своим элементам по ссылке. Они, насколько это возможно, инкапсулируют расположение своих элементов, обернув их в абстракцию. Преимущества такого подхода – обычные плюсы сокрытия информации: у контейнера появляется свобода выбора наилучшей стратегии хранения его элементов. Простой пример – определение контейнера, содержащего объекты типа bool. Если бы контейнер был обязан предоставлять доступ к ref bool, ему пришлось бы хранить каждое значение по отдельному адресу. Если же контейнер вправе скрывать адреса, то он может сохранить восемь логических значений в одном байте.

Другой пример: для некоторых контейнеров доступ к данным неотделим от их изменения. Представим разреженный массив. Разреженные массивы могут фиктивно содержать миллионы элементов, из которых лишь горстка ненулевые, что позволяет разреженным массивам применять стратегии хранения, экономичные в плане занимаемого места. А теперь рассмотрим следующий код:

```
SparseArray!double a;

a[8] += 2;
```

Что должен предпринять массив, зависит как от его текущего содержимого, так и от новых данных: если ячейка a[8] не была ранее заполнена, то создать ячейку со значением 2; если ячейка была заполнена значением -2, удалить эту ячейку, поскольку ее новым значением будет ноль, а такие значения явно не сохраняются; если же ячейка содержала нечто помимо -2, выполнить сложение и записать полученное значение обратно в ячейку. Реализовать эти действия или хотя бы большинство

из них невозможно, если требуется, чтобы метод opIndex возвращал ссылку.

12.7. Перегрузка операторов среза

Массивы D предоставляют операторы среза a[] и a[b_1 .. b_2] (см. раздел 4.1.3). Оба эти оператора могут быть перегружены пользовательскими типами. Компилятор выполняет снижение, примерно как в случае оператора индексации.

Если нет никакого присваивания, компилятор переписывает a[] в виде a.opSlice(), а a[b_1 .. b_2] – в виде a.opSlice(b_1, b_2).

Снижения для операций со срезами делаются по образцу снижений для соответствующих операций, определенных для массивов. Во всех именах методов Index заменяется на Slice: *‹оп›* a[] снижается до a.opSliceUnary!"*‹оп›*"(), *‹оп›* a[b_1 .. b_2] превращается в a.opSliceUnary!"*‹оп›*"(b_1, b_2), a[] = c – в a.opSliceAssign(c), a[b_1 .. b_2] = c – в a.opSliceAssign(c, b_1, b_2), a[] *‹оп›*= c – в a.opSliceOpAssign!"*‹оп›*"(c), и наконец, a[b_1 .. b_2] *‹оп›*= c – в a.opSliceOpAssign!"*‹оп›*"(c, b_1, b_2).

12.8. Оператор $

В случае встроенных массивов язык D позволяет внутри индексных выражений и среза обозначить длину массива идентификатором $. Например, выражение a[0 .. $ - 1] выбирает все элементы встроенного массива a кроме последнего.

Хотя этот оператор с виду довольно скромен, оказалось, что $ сильно повышает и без того хорошее настроение программиста на D. С другой стороны, если бы оператор $ был «волшебным» и не допускал перегрузку, это бы неизменно раздражало, еще раз подтверждая, что встроенные типы должны лишь изредка обладать возможностями, недоступными пользовательским типам.

Для пользовательских типов оператор $ может быть перегружен так:

- для выражения a[*‹выраж›*], где a имеет пользовательский тип: если в *‹выраж›* встречается $, оно переписывается как a.opDollar(). Замена одна и та же независимо от присваивания этого выражения;
- для выражения a[*‹выраж*$_1$*›*, ..., *‹выраж*$_k$*›*]: если в *‹выраж*$_1$*›* встречается $, оно переписывается как a.opDollar!(i)();
- для выражения a[*‹выраж*$_1$*›* .. *‹выраж*$_2$*›*]: если в *‹выраж*$_1$*›* или *‹выраж*$_2$*›* встречается $, оно переписывается как a.opDollar().

Если a – результат некоторого выражения, это выражение вычисляется только один раз.

12.9. Перегрузка foreach

Пользовательские типы могут существенным образом определять, как цикл просмотра будет с ними работать. Это огромное благо для типов, моделирующих коллекции, диапазоны, потоки и другие сущности, элементы которых можно перебирать. Более того, дела обстоят еще лучше: есть целых два независимых способа организовать перегрузку, со своими плюсами и минусами.

12.9.1. foreach с примитивами перебора

Первый способ определить, как цикл foreach должен работать с вашим типом (структурой или классом), заключается в определении трех примитивов перебора: свойства empty типа bool, сообщающего, остались ли еще непросмотренные элементы, свойства front, возвращающего текущий просматриваемый элемент, и метода popFront()[1], осуществляющего переход к следующему элементу. Вот типичная реализация этих трех примитивов:

```
struct SimpleList(T) {
private:
   struct Node {
      T _payload;
      Node * _next;
   }
   Node * _root;
public:
   @property bool empty() { return !_root; }
   @property ref T front() { return _root._payload; }
   void popFront() { _root = _root._next; }
   ...
}
```

Имея такое определение, организовать перебор элементов списка проще простого:

```
void process(SimpleList!int lst) {
   foreach (value; lst) {
       .. // Использовать значение типа int
   }
}
```

Компилятор заменяет управляющий код foreach соответствующим циклом for, более неповоротливым, но мелкоструктурным аналогом, который и использует три рассмотренные примитива:

```
void process(SimpleList!int lst) {
   for (auto __c = lst; !__c.empty; __c.popFront()) {
```

[1] Для перегрузки foreach_reverse служат примитивы popBack и back аналогичного назначения. – *Прим. науч. ред.*

```
        auto value = __c.front;
            // Использовать значение типа int
    }
}
```

Если вы снабдите аргумент value ключевым словом ref, компилятор заменит все обращения к value в теле цикла обращениями к свойству __c.front. Таким образом, вы получаете возможность изменять элементы списка напрямую. Конечно, и само ваше свойство front должно возвращать ссылку, иначе попытки использовать его как l-значение породят ошибки.

Последнее, но не менее важное: если просматриваемый объект предоставляет оператор среза без аргументов lst[], __c инициализируется выражением lst[], а не lst. Это делается для того, чтобы разрешить «извлечь» из контейнера средства перебора, не требуя определения трех примитивов перебора.

12.9.2. foreach с внутренним перебором

Примитивы из предыдущего раздела образуют интерфейс перебора, который клиентский код может использовать, как заблагорассудится. Но иногда лучше использовать *внутренний перебор*, когда просматриваемая сущность полностью управляет процессом перебора и выполняет тело цикла самостоятельно. Такое перекладывание ответственности часто может быть полезно, например, если полный просмотр коллекции предпочтительнее выполнять рекурсивно (как в случае с деревьями).

Чтобы организовать цикл foreach с внутренним перебором, для вашей структуры или класса нужно определить метод opApply[1]. Например:

```
import std.stdio;

class SimpleTree(T) {
private:
   T _payload;
   SimpleTree _left, _right;

public:
   this(T payload) {
      _payload = payload;
   }

   // Обход дерева в глубину
   int opApply(int delegate(ref T) dg) {
      auto result = dg(_payload);
      if (result) return result;
```

[1] Существует также оператор opApplyReverse, предназначенный для перегрузки foreach_reverse и действующий аналогично opApply для foreach. – *Прим. науч. ред.*

```
        if (_left) {
            result = _left.opApply(dg);
            if (result) return result;
        }
        if (_right) {
            result = _right.opApply(dg);
            if (result) return result;
        }
        return 0;
    }
}

void main() {
    auto obj = new SimpleTree!int(1);
    obj._left = new SimpleTree!int(5);
    obj._right = new SimpleTree!int(42);
    obj._right._left = new SimpleTree!int(50);
    obj._right._right = new SimpleTree!int(100);
    foreach (i; obj) {
        writeln(i);
    }
}
```

Эта программа выполняет обход дерева в глубину и выводит:

```
1
5
42
50
100
```

Компилятор упаковывает тело цикла (в данном случае { writeln(i); }) в делегат и передает его методу opApply. Компилятор организует исполнение программы так, что код, выполняющий выход из цикла с помощью инструкции break, преждевременно возвращает 1 в качестве результата делегата, отсюда и манипуляции с result внутри opApply.

Зная все это, читать код метода opApply действительно легко: сначала тело цикла применяется к корневому узлу, а затем рекурсивно к левому и правому узлам. Простота реализации действительно имеет значение. Если вы попробуете реализовать просмотр узлов дерева с помощью примитивов empty, front и popFront, задача сильно усложнится. Так происходит потому, что в методе opApply состояние итерации формируется неявно благодаря стеку вызовов. А при использовании трех примитивов перебора вам придется управлять этим состоянием явно.

Упомянем еще одну достойную внимания деталь во взаимодействии foreach и opApply. Переменная i, используемая в цикле, становится частью типа делегата. К счастью, на тип этой переменной и даже на число привязываемых к делегату переменных, задействованных в foreach, ограничения не налагаются – все поддается настройке. Если вы опреде-

лите метод opApply так, что он будет принимать делегат с двумя аргументами, то сможете использовать цикл foreach следующего вида:

```
// Вызывает метод object.opApply(delegate int(ref K k, ref V v){...})
foreach (k, v; object) {

}
```

На самом деле, просмотр ключей и значений встроенных ассоциативных массивов реализован именно с помощью opApply. Для любого ассоциативного массива типа V[K] справедливо, что делегат, принимаемый методом opApply, ожидает в качестве параметров значения типов V и K.

12.10. Определение перегруженных операторов в классах

Большинство рассмотренных замен включали вызовы методов с параметрами времени компиляции, таких как opBinary(string)(T). Такие методы очень хорошо работают как внутри классов, так и внутри структур. Единственная проблема в том, что методы с параметрами времени компиляции неявно неизменяемы, и их нельзя переопределить, так что для определения класса или интерфейса с переопределяемыми элементами может потребоваться ряд дополнительных шагов. Простейшее решение – написать, к примеру, метод opBinary, так чтобы он проталкивал выполнение операции далее в обычный метод, который можно переопределить:

```
class A {
   // Метод, не допускающий переопределение
   A opBinary(string op)(A rhs) {
      // Протолкнуть в функцию, допускающую переопределение
      return opBinary(op, rhs);
   }
   // Переопределяемый метод, управляется строкой во время исполнения
   A opBinary(string op, A rhs) {
      switch (op) {
         case "+":
               // Реализовать сложение
            break;
         case "-":
               // Реализовать вычитание
            break;
         ..
      }
   }
}
```

Такой подход позволяет решить поставленную задачу, но не оптимально, ведь оператор проверяется во время исполнения – действие, которое может быть выполнено во время компиляции. Следующее решение по-

зволяет исключить излишние затраты по времени за счет переноса проверки внутрь обобщенной версии метода opBinary:

```
class A {
   // Метод, не допускающий переопределение
   A opBinary(string op)(A rhs) {
      // Протолкнуть в функцию, допускающую переопределение
      static if (op == "+") {
         return opAdd(rhs);
      } else static if (op == "-") {
         return opSubtract(rhs);
      }
   }
   // Переопределяемые методы
   A opAdd(A rhs) {
         // Реализовать сложение
   }
   A opSubtract(A rhs) {
      .. // Реализовать вычитание
   }

}
```

На этот раз каждому оператору соответствует свой метод. Вы, разумеется, вправе выбрать операторы для перегрузки и способы их группирования, соответствующие вашему случаю.

12.11. Кое-что из другой оперы: opDispatch

Пожалуй, самая интересная из замен, открывающая максимум возможностей, – это замена с участием метода opDispatch. Именно она позволяет D встать в один ряд с гораздо более динамическими языками.

Если некоторый тип T определяет метод opDispatch, компилятор переписывает выражение

```
a.fun(‹arг1›, ..., ‹argk›)
```

как

```
a.opDispatch!"fun"(‹arг1›,       ‹argk›)
```

для всех методов fun, которые должны были бы присутствовать, но не определены, то есть для всех вызовов, которые бы иначе вызвали ошибку «метод не определен».

Определение opDispatch может реализовывать много очень интересных задумок разной степени динамичности. Рассмотрим пример метода opDispatch, реализующего подчинение альтернативному соглашению именования методов класса. Для начала объявим простую функцию, преобразующую идентификатор такого_вида в его альтернативу «в стиле верблюда» (camel-case) такогоВида:

```
import std.ctype;

string underscoresToCamelCase(string sym) {
   string result;
   bool makeUpper;
   foreach (c; sym) {
      if (c == '_') {
         makeUpper = true;
      } else {
         if (makeUpper) {
            result ~= toupper(c);
            makeUpper = false;
         } else {
            result ~= c;
         }
      }
   }
   return result;
}

unittest {
   assert(underscoresToCamelCase("здравствуй_мир") == "здравствуйМир");
   assert(underscoresToCamelCase("_a") == "A");
   assert(underscoresToCamelCase("abc") == "abc");
   assert(underscoresToCamelCase("a_bc_d_") == "aBcD");
}
```

Вооружившись функцией underscoresToCamelCase, **можно легко определить для некоторого класса метод** opDispatch, **заставляющий этот класс принимать вызовы** a.метод_такого_вида() **и автоматически перенаправлять эти обращения к методам** a.методТакогоВида() **— и все это во время компиляции.**

```
class A {
   auto opDispatch(string m, Args...)(Args args) {
      return mixin("this."~underscoresToCamelCase(m)~"(args)");
   }
   int doSomethingCool(int x, int y) {
      ...
      return 0;
   }
}

unittest {
   auto a = new A;
   a.doSomethingCool(5, 6);    // Вызов напрямую
   a.do_something_cool(5, 6); // Тот же вызов, но через
                              // посредника opDispatch
}
```

Второй вызов не относится ни к одному из методов класса A, **так что он перенаправляется в метод** opDispatch **через вызов** a.opDispatch!"do_some-

thing_cool"(5, 6). opDispatch, в свою очередь, генерирует строку "this.doSomethingCool(args)", а затем компилирует ее с помощью выражения mixin. Учитывая, что с переменной args связана пара аргументов 5, 6, вызов mixin в итоге сменяется вызовом a.doSomethingCool(5, 6) — старое доброе перенаправление в своем лучшем проявлении. Миссия выполнена!

12.11.1. Динамическое диспетчирование с opDispatch

Хотя, конечно, интересно использовать opDispatch в разнообразных проделках времени компиляции, реально интересные приложения требуют динамичности. Динамические языки, такие как JavaScript или Smalltalk, позволяют присоединять к объектам методы во время исполнения. Попробуем сделать нечто подобное на D: определим класс Dynamic, позволяющий динамически добавлять, удалять и вызывать методы.

Во-первых, для таких динамических методов придется определить сигнатуру времени исполнения. Здесь нам поможет тип Variant из модуля std.variant. Это мастер на все руки: объект типа Variant может содержать практически любое значение. Такое свойство делает Variant идеальным кандидатом на роль типа параметра и возвращаемого значения динамического метода. Итак, определим сигнатуру такого динамического метода в виде делегата, который в качестве первого аргумента (играющего роль this) принимает Dynamic, а вместо остальных аргументов — массив элементов типа Variant, и возвращает результат типа Variant.

```
import std.variant;

alias Variant delegate(Dynamic self, Variant[] args...) DynMethod;
```

Благодаря ... можно вызывать DynMethod с любым количеством аргументов с уверенностью, что компилятор упакует их в массив. А теперь определим класс Dynamic, который, как и обещано, позволит манипулировать методами во время исполнения. Чтобы обеспечить такие возможности, Dynamic определяет ассоциативный массив, отображающий строки на элементы типа DynMethod:

```
class Dynamic {
   private DynMethod[string] methods;
   void addMethod(string name, DynMethod m) {
      methods[name] = m;
   }
   void removeMethod(string name) {
      methods.remove(name);
   }
   // Динамическое диспетчирование вызова метода
   Variant call(string methodName, Variant[] args...) {
      return methods[methodName](this, args);
   }
   // Предоставить синтаксический сахар с помощью opDispatch
   Variant opDispatch(string m, Args...)(Args args) {
      Variant[] packedArgs = new Variant[args.length];
```

```
        foreach (i, arg; args) {
            packedArgs[i] = Variant(arg);
        }
        return call(m, args);
    }
}
```

Посмотрим на `Dynamic` в действии:

```
unittest {
    auto obj = new Dynamic;
    obj.addMethod("sayHello"
        delegate Variant(Dynamic, Variant[]...) {
            writeln("Здравствуй, мир!");
            return Variant();
        });
    obj.sayHello(); // Печатает "Здравствуй, мир!"
}
```

Поскольку все методы должны соответствовать одной и той же сигнатуре, добавление метода не обходится без некоторого синтаксического шума. В этом примере довольно много незадействованных элементов: добавляемый делегат не использует ни один из своих параметров и возвращает результат, не представляющий никакого интереса. Зато синтаксис вызова очень прозрачен. Это важно, так как обычно методы добавляются редко, а вызываются часто. Усовершенствовать класс `Dynamic` можно разными путями. Например, можно определить информационную функцию `getMethodInfo(string)`, возвращающую для заданного метода число его параметров и их типы.

Заметим, что в данном случае приходится идти на уступки, обычные для решения о статическом или динамическом выполнении действий. Чем больше вы делаете во время исполнения, тем чаще требуется соответствовать общим форматам данных (`Variant` в нашем примере) и идти на компромисс, жертвуя быстродействием (например, из-за поиска имен методов во время исполнения). Взамен вы получаете возросшую гибкость: можно как угодно манипулировать определениями классов во время исполнения, определять отношения динамического наследования, взаимодействовать со скриптовыми языками, определять скрипты для собственных объектов и еще много чего.

12.12. Итоги и справочник

Пользовательские типы могут перегружать большинство операторов. Есть несколько исключений, таких как «запятая» `,`, логическая конъюнкция `&&`, логическая дизъюнкция `||`, проверка на идентичность `is`, тернарный оператор `?:`, а также унарные операторы получения адреса `&` и `typeid`. Было решено, что перегрузка этих операторов добавит скорее путаницы, чем гибкости.

Кстати, о путанице. Заметим, что перегрузка операторов – это мощный инструмент, к которому прилагается инструкция с предупреждением той же мощности. В языке D лучший совет для вас: не используйте операторы в экзотических целях, вроде определения целых встроенных предметно-ориентированных языков (Domain-Specific Embedded Language, DSEL). Если желаете определять встроенные предметно-ориентированные языки, то для этой цели лучше всего подойдут строки и выражение `mixin` (см. раздел 2.3.4.2) с вычислением функций на этапе компиляции (см. раздел 5.12). Эти средства позволяют выполнить синтаксический разбор входных конструкций на DSEL, представленных в виде строки времени компиляции, а затем сгенерировать соответствующий код на D. Такой подход требует больше труда, но пользователи вашей библиотеки это оценят.

Определение `opDispatch` открывает новые горизонты, но это средство также нужно использовать с умом. Чрезмерная динамичность может снизить быстродействие программы за счет лишних манипуляций и ослабить проверку типов (например, не стоит забывать, что если в предыдущем фрагменте кода вместо `a.helloWorld()` написать `a.heloWorld()`, код все равно скомпилируется, а ошибка проявится лишь во время исполнения).

В табл. 12.1 в сжатой форме представлена информация из этой главы. Используйте эту таблицу как шпаргалку, когда будете перегружать операторы для собственных типов.

Таблица 12.1. Перегруженные операторы

Выражение	Переписывается как...
*‹оп›*a, где *‹оп›* ∈ {+, -, ~, *, ++, --}	a.opUnary!"*‹оп›*"()
a++	((ref x) {auto t=x; ++x; return t;})(a)
a--	((ref x) {auto t=x; --x; return t;})(a)
cast(T) a	a.opCast!(T)()
a ? *‹выраж$_1$›* : *‹выраж$_2$›*	cast(bool) a ? *‹выраж$_1$›* : *‹выраж$_2$›*
if (a) *‹инстр›*	if (cast(bool) a) *‹инстр›*
a *‹оп›* b, где *‹оп›* ∈ {+, -, *, /, %, &, |, ^, <<, >>, >>>, ~, in}	a.opBinary!"*‹оп›*"(b) **или** b.opBinaryRight!"*‹оп›*"(a)
a == b	**Если** a **и** b **– экземпляры классов:** object.opEquals(a, b) **(см. раздел 6.8.3). Иначе если** a **и** b **имеют один тип:** a.opEquals(b). **Иначе единственное выражение из** a.opEquals(b) **и** b.opEquals(a), **которое компилируется**
a != b	!(a == b), **затем действовать по предыдущему алгоритму**

Таблица 12.1 (продолжение)

Выражение	Переписывается как...
a < b	a.opCmp(b) < 0 **или** b.opCmp(a) > 0
a <= b	a.opCmp(b) <= 0 **или** b.opCmp(a) >= 0
a > b	a.opCmp(b) > 0 **или** b.opCmp(a) < 0
a >= b	a.opCmp(b) >= 0 **или** b.opCmp(a) <= 0
a = b	a.opAssign(b)
a *‹оп›*= b, **где** *‹оп›* ∈ {+, -, *, /, %, &, |, ^, <<, >>, >>>, ~}	a.opOpAssign!"*‹оп›*"(b)
a[b_1, b_2, ..., b_k]	a.opIndex(b_1, b_2, ..., b_k)
a[b_1, b_2, ..., b_k] = c	a.opIndexAssign(c, b_1, b_2, ..., b_k)
*‹оп›*a[b_1, b_2, ..., b_k], **где** *‹оп›* ∈ {++, --}	a.opIndexUnary(b_1, b_2, ..., b_k)
a[b_1, b_2, ..., b_k] *‹оп›*= c, **где** *‹оп›* ∈ {+, -, *, /, %, &, |, ^, <<, >>, >>>, ~}	a.opIndexOpAssign!"*‹оп›*"(c, b_1, b_2, ..., b_k)
a[b_1 .. b_2]	a.opSlice(b_1 .. b_2)
‹оп› a[b_1 .. b_2]	a.opSliceUnary!"*‹оп›*"(b_1, b_2)
a[] = c	a.opSliceAssign(c)
a[b_1 .. b_2] = c	a.opSliceAssign(c, b_1, b_2)
a[] *‹оп›*= c	a.opSliceOpAssign!"*‹оп›*"(c)
a[b_1 .. b_2] *‹оп›*= c	a.opSliceOpAssign!"*‹оп›*"(c, b_1, b_2)

13

Параллельные вычисления

Благодаря сложившейся обстановке в индустрии аппаратного обеспечения качественно изменился способ доступа к вычислительным ресурсам, которые, в свою очередь, требуют основательного пересмотра техники вычислений и применяемых языковых абстракций. Сегодня широко распространены параллельные вычисления, и программное обеспечение должно научиться извлекать из этого пользу.

Несмотря на то что индустрия программного обеспечения в целом еще не выработала окончательные ответы на вопросы, поставленные революцией в области параллельных вычислений, молодость D позволила его создателям, не связанным ни устаревшими концепциями прошлого, ни огромным наследством базового кода, принять компетентные решения относительно параллелизма. Главное отличие подхода D от стандарта поддерживающих параллелизм императивных языков – в том, что он не поощряет разделение данных между потоками; по умолчанию параллельные потоки фактически изолированы друг от друга с помощью механизмов языка. Разделение данных разрешено, но лишь в ограниченной управляемой форме, чтобы компилятор мог предоставлять основательные глобальные гарантии.

В то же время D, оставаясь в душе языком для системного программирования, разрешает применять ряд низкоуровневых, неконтролируемых механизмов параллельных вычислений. (При этом в безопасных программах некоторые из этих механизмов использовать запрещено.)

Вот краткий обзор уровней параллелизма, предлагаемых языком D:

- Передовой подход к параллельным вычислениям заключается в использовании изолированных потоков или процессов, взаимодействующих с помощью сообщений. Эта парадигма, называемая *обменом сообщениями* (*message passing*), позволяет создавать безопасные модульные программы, легкие для понимания и сопровождения.

Обмен сообщениями успешно применяется в разнообразных языках и библиотеках. Раньше обмен сообщениями был медленнее подходов, основанных на разделении памяти, поэтому он и не стал общепринятым, но за последнее время здесь многое бесповоротно изменилось. Параллельные программы на D используют обмен сообщениями – парадигму, ориентированную на всестороннюю инфраструктурную поддержку.

- D также поддерживает старомодную синхронизацию на основе критических участков, защищенных мьютексами и флагами событий. В последнее время этот подход к организации параллельных вычислений подвергается серьезной критике за недостаточную масштабируемость для настоящих и будущих параллельных архитектур. D строго управляет разделением данных, ограничивая возможности программирования с применением блокировок. На первый взгляд это ограничение может показаться суровым, но оно избавляет основанный на блокировках код от его злейшего врага – низкоуровневых гонок за данными (ситуаций состязания). При этом разделение данных остается наиболее эффективным средством передачи больших объемов данных между потоками, так что пренебрегать им не стоит.
- По традиции языков системного уровня программы на D, не имеющие атрибута `@safe`, могут посредством приведений достигать беспрепятственного разделения данных. За корректность таких программ в основном отвечаете вы.
- Если вам мало предыдущего уровня, конструкция `asm` позволяет получить полный контроль над машинными ресурсами. Для еще более низкоуровневого контроля потребуются микропаяльник и очень, очень верная рука.

Прежде чем с головой окунуться во все это, отвлечемся ненадолго, чтобы поближе присмотреться к тем аппаратным усовершенствованиям, которые потрясли мир.

13.1. Революция в области параллельных вычислений

Что касается параллельных вычислений, то для них сейчас времена поинтереснее, чем когда-либо. Это времена, когда и хорошие, и плохие новости вписываются в общую панораму компромиссов, противоборств и тенденций.

Хорошие новости в том, что степень интеграции все еще растет по закону Мура[1]; судя по тому, что нам уже известно, и по тому, что мы сегодня можем предположить, это продлится как минимум лет десять после

[1] Число транзисторов на кристалл будет увеличиваться вдвое каждые 24 месяца. – *Прим. пер.*

выхода этой книги. Курс на миниатюризацию означает рост плотности вычислительной мощности пропорционально числу совместно работающих транзисторов на единицу площади. Все ближе друг к другу компоненты, все короче соединения, а это означает повышение скорости локальной связности – золотое дно в плане быстродействия.

К сожалению, отдельные выводы, начинающиеся со слов «к сожалению», умеряют энтузиазм по поводу возросшей вычислительной плотности. Во-первых, существует не только локальная связность – она формируется в иерархию [16]: тесно связанные компоненты образуют блоки, которые должны связываться с другими блоками, образуя блоки большего размера. В свою очередь, блоки большего размера также соединяются с другими блоками большего размера, образуя функциональные блоки еще большего размера, и т. д. На своем уровне связности такие блоки остаются «далеки» друг от друга. Хуже того, возросшая сложность каждого блока увеличивает сложность связей между блоками, что реализуется путем уменьшения толщины проводов и расстояния между ними. Это означает рост сопротивления, электроемкости и перекрестных помех. Перекрестные помехи – это способность сигнала из одного провода распространяться на соседние провода посредством (в данном случае) электромагнитного поля. На высоких частотах провод – практически антенна, и помехи становятся настолько невыносимыми, что сегодня параллельные соединения все чаще заменяют последовательными (своего рода феномен нелогичности, заметный на всех уровнях: USB заменил параллельный порт, в качестве интерфейса накопителей данных SATA заменил PATA, а в подсистемах памяти последовательные шины заменяют параллельные, и все из-за перекрестных помех. Где те золотые деньки, когда параллельное было быстрее, а последовательное медленнее?).

Кроме того, растет разрыв в производительности между вычислительными элементами и памятью. В то время как плотность памяти, как и ожидалось, увеличивается в соответствии с общей степенью интеграции, скорость доступа к ней все больше отстает от скорости вычислений из-за множества разнообразных физических, технологических и рыночных факторов [22]. В настоящее время неясно, что поможет существенно сократить этот разрыв в быстродействии, и он лишь растет. Тысячи тактов могут отделять процессор от слова в памяти; а ведь еще несколько лет назад можно было купить микросхемы памяти «с нулевым временем ожидания», обращение к которым осуществлялось за один такт.

Из-за широкого спектра архитектур памяти, представляющих собой различные компромиссные решения относительно плотности, цены и скорости, повысилась и изощренность иерархий памяти; обращение к единственному слову памяти превратилось в детективное расследование с опросом нескольких уровней кэша, начиная с драгоценного статического ОЗУ прямо на микросхеме и порой проходя весь путь до массовой памяти. Возможна и противоположная ситуация: копии указан-

ных данных могут располагаться во множестве мест по всей иерархии. Это, в свою очередь, тоже влияет на модели программирования. Мы больше не можем позволить себе представлять память большим монолитом, удобным для разделения всеми процессами системы: наличие кэшей провоцирует рост локального трафика в памяти, превращая разделяемые данные в иллюзию, которую все труднее сопровождать [37].

К последним сенсационным известиям относится то, что скорость света упрямо решила оставаться неизменной (`immutable`, если хотите) – около 300000000 метров в секунду. Скорость же света в оксиде кремния (соответствующая скорости распространения сигнала внутри современных микросхем) составляет примерно половину этого значения, причем достижимая сегодня скорость переноса самих данных существенно ниже этого теоретического предела. Это означает больше проблем с глобальной взаимосвязанностью на высоких частотах. Если бы у нас была микросхема с частотой 10 ГГц, то простое перемещение бита с одного на другой конец этого чипа шириной 4,5 см (по сути, вообще без вычислений) в идеальных условиях занимало бы три такта.

Словом, наступает век процессоров очень высокой плотности и гигантской вычислительной мощности, при этом все более изолированных и труднодоступных, которые сложно использовать из-за ограничений взаимосвязности, скорости распространения сигнала и быстроты доступа к памяти.

Компьютерная индустрия, естественно, обходит эти преграды. Одним из феноменов стало резкое сокращение размеров и энергии, требуемых для заданной вычислительной мощности; всего лишь пять лет назад уровень технологии не позволял достичь компактности и возможностей КПК, без которых сегодня мы как без рук. При этом традиционные компьютеры, пытающиеся повысить вычислительную мощность при тех же размерах, представляют все меньший интерес. Производители микросхем для них уже не борются за повышение тактовой частоты, предлагая взамен вычислительную мощность в уже знакомой упаковке: несколько одинаковых центральных процессоров, соединенных шинами друг с другом и с памятью. Так что спустя каких-то несколько лет отвечать за разгон компьютеров будут не электронщики, а в основном программисты. Вариант «побольше процессоров» может показаться довольно заманчивым, но типовым задачам настольного компьютера не под силу эффективно использовать и восемь процессоров. В будущем предполагается экспоненциальный рост числа доступных процессоров до десятков, сотен и тысяч. При разгоне единственной программы программистам придется очень много потрудиться, чтобы продуктивно использовать *все* эти процессоры.

Из-за разных технологических и человеческих факторов в компьютерной индустрии постоянно случаются подвижки и сотрясения, но на этот раз мы, кажется, дошли до точки. С недавних пор взять отпуск, чтобы увеличить скорость работы программы, – уже не вариант. Это

возмутительно. Это подрыв устоев. Это революция в области параллельных вычислений.

13.2. Краткая история механизмов разделения данных

Один из аспектов перемен в компьютерной индустрии – внезапность, с какой сегодня меняются модели обработки данных и параллелизма, особенно на фоне темпа развития языков и парадигм программирования. Чтобы язык и связанные с ним стили отпечатались в сознании сообщества программистов, нужны годы, даже десятки лет, а в области параллелизма начиная с 2000-х все меняется в геометрической прогрессии.

Например, наше прошлогоднее понимание основ параллелизма[1] тяготело к разделению данных, порожденному мейнфреймами 1960-х. Тогда процессорное время было настолько дорогим, что повысить общую эффективность использования процессора можно было, только разделяя его между множеством программ, управляемых множеством операторов. *Процесс* определялся и определяется как совокупность состояния и ресурсов исполняющейся программы. Процессор (центральное процессорное устройство, ЦПУ) реализует разделение времени с помощью планировщика задач и прерываний таймера. По каждому прерыванию таймера планировщик решает, какому процессу предоставить ЦПУ на следующий квант времени, создавая таким образом иллюзию одновременного исполнения нескольких процессов, хотя на самом деле все они используют одно и то же ЦПУ.

Чтобы ошибочные процессы не повредили друг другу и коду операционной системы, была введена *аппаратная защита памяти*. Для надежной изоляции процессов в современных системах защиту памяти сочетают с *виртуализацией памяти*: каждый процесс считает память машины «своей собственностью», хотя на самом деле все взаимодействие между процессом и памятью, а также изоляцию процессов друг от друга берет на себя уровень-посредник, транслирующий логические адреса (так видит память процесс) в физические (так обращается к памяти машина). Хорошие новости в том, что процессы, вышедшие из-под контроля, могут навредить только себе, но не другим процессам и не ядру операционной системы. Новости похуже в том, что каждое переключение задач требует потенциально дорогой смены адресных пространств процессов, не говоря о том, что каждый процесс при переключении на него «просыпается» с амнезией кэша, поскольку глобальный кэш обычно используется всеми процессами. Так и появились *потоки* (*threads*).

---

[1] Далее речь идет о параллельных вычислениях в целом и не рассматриваются распараллеливание операций над векторами и другие специализированные параллельные функции ядра.

Поток – это процесс, не владеющий информацией о том, как транслировать адреса; это чистый контекст исполнения: состояние процессора плюс стек. Несколько потоков разделяют адресное пространство процесса, то есть порождать потоки и переключаться между ними относительно дешево, и они могут с легкостью и без особых затрат разделять данные друг с другом. Разделение памяти между потоками, запущенными на одном ЦПУ, осуществляется настолько прямолинейно, насколько это возможно: один поток пишет, другой читает. При использовании техники разделения времени порядок записи данных, естественно, совпадает с порядком, в котором эти записи будут видны другим потокам. Поддержку более высокоуровневых инвариантов данных обеспечивают механизмы блокировки, например критические секции, защищенные с помощью примитивов синхронизации (таких как семафоры и мьютексы). В последние годы XX века то, что можно назвать «классическим» многопоточным программированием (которое характеризуется разделяемым адресным пространством, простыми правилами видимости изменений и синхронизацией на мьютексах), обросло массой наблюдений, народных мудростей и анекдотов. Существовали и другие модели организации параллельных вычислений, но на большинстве машин применялась классическая многопоточность.

Основные императивные языки наших дней (такие как C, C++, Java) развивались в век классической многопоточности – в старые добрые времена простых архитектур памяти, понятных примитивов взаимоблокировки и разделения данных без изысков. Языки, естественно, моделировали реалии аппаратного обеспечения того времени (когда подразумевалось, что потоки разделяют одну и ту же область памяти) и включали соответствующие средства. В конце концов само определение многопоточности подразумевает, что все потоки, в отличие от процессов операционной системы, разделяют одно общее адресное пространство. Кроме того, API для реализации обмена сообщениями (например, спецификация MPI [29]) были доступны лишь в форме библиотек, изначально созданных для специализированного дорогостоящего аппаратного обеспечения, такого как кластеры (супер)компьютеров.

Тогда еще только зарождающиеся функциональные языки заняли принципиальную позицию, основанную на математической чистоте: «Мы не заинтересованы в моделировании аппаратного обеспечения, – сказали они. – Нам хотелось бы моделировать математику». А в математике редко что-то меняется, математические результаты инвариантны во времени, что делает математические вычисления идеальным кандидатом для распараллеливания. (Только представьте, как первые программисты – вчерашние математики, услышав о параллельных вычислениях, чешут затылки, восклицая: «Минуточку!..») Функциональные программисты убеждены, что такая модель вычислений поощряет неупорядоченное, параллельное выполнение, однако до недавнего времени эта возможность являлась скорее потенциальной энергией, чем достигнутой целью.

Наконец был разработан язык Erlang. Он начал свой путь в конце 1980-х как предметно-ориентированный встроенный язык приложений для телефонии. Предметная область, предполагая десятки тысяч программ, одновременно запущенных на одной машине, заставляла отдать предпочтение обмену сообщениями, когда информация передается в стиле «выстрелил – забыл». Аппаратное обеспечение и операционные системы по большей части не были оптимизированы для таких нагрузок, но Erlang изначально запускался на специализированной платформе. В результате получился язык, оригинальным образом сочетающий нечистый функциональный стиль, серьезные возможности для параллельных вычислений и стойкое предпочтение обмена сообщениями (никакого разделения памяти!).

Перенесемся в 2010-е. Сегодня даже у средних машин больше одного процессора, а главная задача десятилетия – уместить на кристалле как можно больше ЦПУ. Отсюда и последствия, самое важное из которых – конец монолитной разделяемой памяти.

С одним разделяемым по времени ЦПУ связана одна подсистема памяти – с буферами, несколькими уровнями кэшей, все по полной программе. Независимо от того, как ЦПУ управляет разделением времени, чтение и запись проходят по одному и тому же маршруту, а потому видение памяти у разных потоков остается когерентным. Несколько взаимосвязанных ЦПУ, напротив, не могут позволить себе разделять подсистему кэша: такой кэш потребовал бы мультипортового доступа (что дорого и слабо масштабируемо), и его было бы трудно разместить в непосредственной близости ко всем ЦПУ сразу. Вот почему практически все современные ЦПУ производятся со своей кэш-памятью, предназначенной лишь для их собственных нужд. Производительность мультипроцессорной системы зависит главным образом от аппаратного обеспечения и протоколов, соединяющих комплексы ЦПУ+кэш.

Несколько кэшей превращают разделение данных между потоками в чертовски сложную задачу. Теперь операции чтения и записи в разных потоках могут затрагивать разные кэши, поэтому сделать так, чтобы один поток делился данными с другим, стало сложнее, чем раньше. На самом деле, этот процесс превращается в своего рода обмен сообщениями[1]: в каждом случае такого разделения между подсистемами кэшей должно иметь место что-то вроде рукопожатия, обеспечивающего попадание разделяемых данных от последнего записавшего потока к читающему потоку, а также в основную память.

Протоколы синхронизации кэшей добавляют к сюжету еще один поворот (хотя и без него все было достаточно лихо закручено): они воспринимают данные только блоками, не предусматривая чтение и запись отдельных слов. То есть общающиеся друг с другом процессы «не помнят»

[1] Что иронично, поскольку во времена классической многопоточности разделение памяти было быстрее, а обмен сообщениями – медленнее.

точный порядок, в котором записывались данные, что приводит к парадоксальному поведению, которое не поддается разумному объяснению и противоречит здравому смыслу: один поток записывает x, а затем y, и в некоторый промежуток времени другой поток видит новое y, но старое x. Такие нарушения причинно-следственных связей слабо вписываются в общую модель классической многопоточности. Даже наиболее сведущим в классической многопоточности программистам невероятно трудно адаптировать свой стиль и шаблоны программирования к новым архитектурам памяти.

Проиллюстрируем скоростные изменения в современных параллельных вычислениях и серьезное влияние разделения данных на подходы языков к параллелизму советом из чудесной книги «Java. Эффективное программирование» издания 2001 года [8, разд. 51, с. 204]:

> «Если есть несколько готовых к исполнению потоков, планировщик потоков определит, какие потоки должны запуститься и на какое время... Лучший способ написать отказоустойчивое, оперативное и переносимое приложение – стараться иметь минимум готовых к исполнению потоков в любой момент времени.»

Сегодняшний читатель сразу же отметит поразительную деталь: здесь не просто говорится об однопроцессорном программировании с многопоточностью на основе разделения времени, но подразумевается единственность процессора, хоть и без явной констатации. Естественно, что в издании 2008 года[1] [9] этот совет был изменен на «стремиться к тому, чтобы среднее число готовых к исполнению потоков было ненамного больше числа процессоров». Любопытно, что даже этот совет, на вид разумный, подразумевает два невысказанных допущения: 1) за счет данных потоки будут сильно связаны друг с другом, что в свою очередь приведет к снижению быстродействия из-за накладных расходов на взаимоблокировки, и 2) число процессоров на машинах, где может запускаться программа, примерно одинаково. И тогда этот совет полностью противоположен тому, что настойчиво повторяется в книге «Программирование на языке Erlang» [5, глава 20, с. 363]:

> **«Используйте много процессоров.** Это важно: мы должны держать свои ЦПУ в занятом состоянии. Все ЦПУ должны быть заняты в каждый момент времени. Легче всего достигнуть этого, имея много процессов[2]. Говоря „много процессов", я имею в виду много по отношению к количеству ЦПУ. Если у нас много процессов, то о занятом состоянии для ЦПУ можно не беспокоиться.»

Какой из этих трех рекомендаций следовать? Как обычно, все зависит от обстоятельств. Первая прекрасно подходит для аппаратного обеспечения

---

[1] Даже заголовок раздела был изменен с «Потоки» на «Параллельные вычисления», чтобы подчеркнуть, что потоки – это не что иное, как одна из моделей параллельных вычислений.

[2] Процессы языка Erlang отличаются от процессов ОС.

2001 года; вторая – для сценариев, характеризующихся интенсивной работой с разделяемыми данными и, следовательно, жестким соперничеством; третья полезна в условиях слабого соперничества и большого количества ЦПУ.

Поддерживать разделение памяти все сложнее, этот подход к организации параллельных вычислений начинает казаться неубедительным, в моду входят функциональность и обмен сообщениями. Неудивительно, что в последние годы растет интерес к Erlang и другим функциональным языкам, удобным для разработки приложений с параллельными вычислениями.

13.3. Смотри, мам, никакого разделения (по умолчанию)

Вследствие последних усовершенствований аппаратного и программного обеспечения D решил отойти от других императивных языков: да, язык D поддерживает потоки, но они не разделяют никакие изменяемые данные по умолчанию – они изолированы друг от друга. Изоляция обеспечивается не аппаратно (как в случае с процессами) и не с помощью проверок времени исполнения; она является естественным следствием устройства системы типов D.

Это решение в духе функциональных языков, которые также стараются запретить любые изменения, а значит, и разделение изменяемых данных. Но есть два различия. Во-первых, программы на D все же могут свободно использовать изменяемые данные – закрыта лишь возможность непреднамеренного обращения к изменяемым данным для других потоков. Во-вторых, «никакого разделения» – это лишь выбор *по умолчанию*, но не *единственный* возможный. Чтобы определить данные как разделяемые между потоками, необходимо уточнить их определение с помощью ключевого слова `shared`. Рассмотрим пример двух простых определений, размещенных в корне модуля:

```
int perThread;
shared int perProcess;
```

В большинстве языков первое определение (или его синтаксический эквивалент) означало бы ввод глобальной переменной, используемой всеми потоками, но в D у переменной `perThread` есть отдельная копия для каждого потока. Второе определение выделяет память лишь под одно значение типа `int`, разделяемое всеми потоками, так что в некотором роде оно ближе (но не идентично) к традиционной глобальной переменной.

Переменная `perThread` сохраняется при помощи средства операционной системы, называемого локальным хранилищем потока (thread-local storage, TLS). Скорость доступа к данным, память под которые выделена в TLS, зависит от реализации компилятора и базовой операционной системы. В общем случае эта скорость лишь незначительно меньше,

скажем, скорости обращения к обычной глобальной переменной в программе на C. В редких случаях, когда эта разница может иметь значение, например в циклах, где делается множество обращений к переменной в TLS, можно загрузить глобальную переменную в стековую.

У такого подхода есть два важных преимущества. Во-первых, языки, по умолчанию использующие разделение, должны тщательно синхронизировать доступ к глобальным данным; для perThread же это необязательно, потому что у каждого потока есть ее локальная копия. Во-вторых, квалификатор shared означает, что и система типов, и программист в курсе, что к переменной perProcess одновременно обращаются многие потоки. В частности, система типов активно защищает разделяемые данные, запрещая использовать их очевидно некорректным образом. D переворачивает традиционные представления с ног на голову: в режиме разделения по умолчанию программист обязан вручную отслеживать, какие данные разделяются, а какие нет, и ведь на самом деле, большинство ошибок, имеющих место при параллельных вычислениях, бывают вызваны чрезмерным или незащищенным разделением данных. В режиме явного разделения программист точно знает, что данные, *не* помеченные квалификатором shared, действительно будут видны только одному потоку. (Для обеспечения такой гарантии значения с пометкой shared проходят дополнительные проверки, до которых мы скоро доберемся.)

Использование разделяемых данных остается делом не для новичков, поскольку, хотя система типов и обеспечивает низкоуровневую когерентность, автоматически обеспечить соблюдение высокоуровневых инвариантов невозможно. Наиболее предпочтительный метод организации безопасного, простого и эффективного обмена информацией между потоками – использовать парадигму *обмена сообщениями*. Обладающие изолированной памятью потоки взаимодействуют, отправляя друг другу асинхронные сообщения, состоящие попросту из совместно упакованных значений D.

Изолированные работники, общающиеся друг с другом с помощью простых каналов коммуникации, – это очень надежный, проверенный временем подход к параллелизму. Язык Erlang и приложения, использующие спецификацию интерфейса передачи сообщений (Message Passing Interface, MPI) [29], применяют его уже давно.

> Намажем мед на пластырь[1]. Даже в языках, использующих разделение данных по умолчанию, хорошая практика программирования фактически предписывает изолировать потоки. Герб Саттер, известный эксперт по параллельным вычислениям, в статье с красноречивым названием «Используйте потоки правильно = изоляция + асинхронные сообщения» [54] пишет: «Потоки – это низкоуровневый инструмент для выражения асинхронных действий. „Приподнимите“ их, установив строгую дисциплину: старайтесь

[1] Подразумевалось обратное от «насыплем соль на рану».

делать их данные локальными, а синхронизацию и обмен информацией организовывать через асинхронные сообщения. Всякий поток, которому нужно получать информацию от других потоков или от людей, должен иметь очередь сообщений (простую очередь FIFO или очередь с приоритетами) и организовывать свою работу, ориентируясь на управляемую событиями помповую магистраль сообщений; замена запутанной логики событийной логикой – чудесный способ улучшить ясность и детерминированность кода.»

Если и есть что-то, чему нас научили десятилетия компьютерных вычислений, так это то, что программирование на базе дисциплины не масштабируется[1]. Но пользователи D могут вздохнуть с облегчением: в данной цитате в основном очень точно изложены тезисы нескольких следующих частей – кроме того, что касается дисциплины.

13.4. Запускаем поток

Для запуска потока воспользуйтесь функцией spawn, как здесь:

```
import std.concurrency, std.stdio;

void main() {
   auto low = 0, high = 100;
   spawn(&fun, low, high);
   foreach (i; low    high) {
      writeln("Основной поток: "  i);
   }
}

void fun(int low, int high) {
   foreach (i; low    high) {
      writeln("Дочерний поток: ", i);
   }
}
```

Функция spawn принимает адрес функции fun и список аргументов ‹a_1›, ‹a_2›, ..., ‹a_n›. Число аргументов n и их типы должны соответствовать сигнатуре функции fun, иными словами, вызов fun(‹a_1›, ‹a_2›, ..., ‹a_n›) должен быть корректным. Эта проверка выполняется во время компиляции. spawn создает новый поток выполнения, который инициирует вызов fun(‹a_1›, ‹a_2›, ..., ‹a_n›), а затем завершает свое выполнение. Конечно же, функция spawn не ждет, когда поток закончит выполняться, – она возвращает управление сразу же после создания потока и передачи ему аргументов (в данном случае двух целых чисел).

Эта программа выводит в стандартный поток вывода в общей сложности 200 строк. Порядок следования этих строк зависит от множества

[1] Речь идет о самом процессе программирования: правила, соблюдение которых компилятор гарантировать не может, люди рано или поздно начнут нарушать (с плачевными последствиями). – *Прим. науч. ред.*

факторов; вполне возможно, что вы увидите 100 строк из основного потока, а затем 100 строк из побочного, в точности противоположную последовательность или некоторое чередование, кажущееся случайным. Однако в одной строке никогда не появится смесь из двух сообщений. Потому что функция writeln определена так, чтобы каждый вызов был атомарен по отношению к потоку вывода. Кроме того, порядок строк, в котором они порождаются каждым из потоков, также будет соблюден.

Даже если выполнение main завершится до окончания выполнения fun в дочернем потоке, программа будет терпеливо ждать того момента, когда завершатся все потоки, и только тогда завершит свое выполнение. Ведь библиотека поддержки времени исполнения подчиняется небольшому протоколу завершения выполнения программ, о котором мы поговорим позже; а пока лишь возьмем на заметку, что даже если main возвращает управление, другие потоки не умирают тут же.

Как и было обещано, у только что созданного потока нет ничего общего с потоком, инициировавшим его. Ну, почти ничего: глобальный дескриптор файла stdout *де факто* разделяется между всеми потоками. Но все же жульничества тут нет: если вы взглянете на реализацию модуля std.stdio, то увидите, что stdout определяется там как глобальная разделяемая переменная. Все грамотно просчитано в системе типов.

13.4.1. Неизменяемое разделение

Какие именно функции можно вызывать из spawn? Установка на отсутствие разделения налагает определенные ограничения: в функцию, запускающую поток (в рассмотренном выше примере это функция fun), параметры можно передавать лишь по значению. Любая передача по ссылке, как явная (в виде параметра с квалификатором ref), так и неявная (например, с помощью массива), должна быть под запретом. Имея в виду это условие, обратимся к новой версии предыдущего примера:

```
import std.concurrency, std.stdio;

void main() {
   auto low = 0, high = 100;
   auto message = "Да, привет #";
   spawn(&fun, message, low, high);
   foreach (i; low    high) {
      writeln("Основной поток: ", message, i);
   }
}

void fun(string text, int low, int high) {
   foreach (i; low    high) {
      writeln("Дочерний поток: "  text, i);
   }
}
```

Переписанный пример идентичен исходному, за исключением того, что печатает еще одну строку. Эта строка создается в основном потоке и передается в дочерний поток без копирования. По сути, содержание message разделяется между потоками. Таким образом, нарушен вышеупомянутый принцип, гласящий, что любое разделение данных должно быть явно помечено ключевым словом shared. Тем не менее код этого примера компилируется и запускается. Что же происходит?

В главе 8 сообщается, что квалификатор immutable предоставляет серьезные гарантии: гарантируется, что помеченное этим ключевым словом значение ни разу не изменится за всю свою жизнь. В той же главе объясняется (см. раздел 8.2), что тип string – это на самом деле псевдоним для типа immutable(char)[]. Наконец, мы знаем, что все споры возникают из-за разделения *изменяемых* данных – пока никто данные не изменяет, можно свободно разделять их, ведь все будут видеть в точности одно и то же. Система типов и инфраструктура потоков в целом признают этот факт, разрешая разделять между потоками все данные, помеченные квалификатором immutable. В частности, можно разделять значения типа string, отдельные знаки которых изменить невозможно. На самом деле, своим появлением в языке квалификатор immutable не в последнюю очередь обязан той помощи, которую он оказывает при разделении структурированных данных между потоками.

13.5. Обмен сообщениями между потоками

Потоки, печатающие сообщения в произвольном порядке, малоинтересны. Изменим наш пример так, чтобы обеспечить работу потоков в тандеме. Добьемся, чтобы они печатали сообщения следующим образом:

```
Основной поток: 0
Дочерний поток: 0
Основной поток: 1
Дочерний поток: 1
   .
Основной поток: 99
Дочерний поток: 99
```

Для этого потребуется определить небольшой протокол взаимодействия двух потоков: основной поток должен отправлять дочернему потоку сообщение «Напечатай это число», а дочерний – отвечать «Печать завершена». Вряд ли здесь имеют место какие-либо параллельные вычисления, но такой пример наглядно объясняет, как организуется взаимодействие в чистом виде. В настоящих приложениях большую часть своего времени потоки должны заниматься полезной работой, а на общение тратить лишь сравнительно малое время.

Начнем с того, что для взаимодействия двух потоков им требуется знать, как обращаться друг к другу. В программе может быть много переговаривающихся потоков, так что средство идентификации необходимо.

Чтобы обратиться к потоку, нужно получить возвращаемый функцией spawn *идентификатор потока* (*thread id*), который с этих пор мы будем неофициально называть «tid». (Тип tid так и называется – Tid.) Дочернему потоку, в свою очередь, также нужен tid, для того чтобы отправить ответ. Это легко организовать, заставив отправителя указать собственный Tid, как пишут адрес отправителя на конверте. Вот этот код:

```
import std.concurrency, std.stdio, std.exception;

void main() {
   auto low = 0, high = 100;
   auto tid = spawn(&writer);
   foreach (i; low    high) {
      writeln("Основной поток: "  i);
      tid.send(thisTid, i);
      enforce(receiveOnly!Tid() == tid);
   }
}

void writer() {
   for (;;) {
      auto msg = receiveOnly!(Tid, int)();
      writeln("Дочерний поток: ", msg[1]);
      msg[0].send(thisTid);
   }
}
```

Теперь функции writer аргументы не нужны: всю необходимую информацию она получает в форме сообщений. Основной поток сохраняет Tid, возвращенный функцией spawn, а затем использует его при вызове метода send. С помощью этого вызова другому потоку отправляются два фрагмента данных: Tid текущего потока (доступ к которому предоставляет глобальная переменная thisTid) и целое число, которое нужно напечатать. Перекинув данные через забор другому потоку, основной поток начинает ждать подтверждение того, что его сообщение получено, в виде вызова receiveOnly. Функции send и receiveOnly работают в тандеме: всякому вызову send в одном потоке ставится в соответствие вызов receiveOnly в другом. В названии receiveOnly присутствует слово «only» (только), потому что receiveOnly принимает только определенные типы, например, инициатор вызова receiveOnly!bool() принимает лишь сообщения в виде логических значений; если другой поток отправляет что-либо другое, receiveOnly порождает исключение типа MessageMismatch.

Предоставим main копаться в цикле foreach и сосредоточимся на функции writer, реализующей вторую часть нашего мини-протокола. writer коротает время в цикле, начинающемся получением сообщения, которое должно состоять из значения типа Tid и значения типа int. Именно это обеспечивает вызов receiveOnly!(Tid, int)(); опять же, если бы основной поток отправил сообщение с каким-либо иным количеством аргументов или с аргументами других типов, receiveOnly прервала бы свое

выполнение по исключению. Как уже говорилось, вызов receiveOnly в теле writer полностью соответствует вызову tid.send(thisTid, i) из main.

Типом msg является Tuple!(Tid, int). В общем случае сообщения со множеством аргументов упаковываются в кортежи, так что одному члену кортежа соответствует один аргумент. Но если сообщение состоит всего из одного значения, лишние движения не нужны, и упаковка в Tuple опускается. Например, receiveOnly!int() возвращает int, а не Tuple!int.

Продолжим разбор writer. Следующая строка, собственно, выполняет печать (запись в консоль). Вспомните, что для кортежа msg выражение msg[0] означает обращение к первому члену кортежа (то есть к Tid), а выражение msg[1] – доступ к его второму члену (к целому числу). Наконец, writer посылает уведомление о том, что завершила запись в консоль, попросту отправляя собственный Tid отправителю предыдущего сообщения – своего рода пустой конверт, лишь подтверждающий личность отправителя. «Да, я получил твое сообщение, – подразумевает пустое письмо, – и принял соответствующие меры. Твоя очередь.» Основной поток не продолжит работу, пока не получит такое уведомление, но как только это произойдет, цикл начнет выполняться дальше.

Отправлять Tid дочернего потока назад в данном случае излишне – хватило бы любой болванки, например int или bool. Однако в общем случае в программе есть много потоков, отправляющих друг другу сообщения, так что самоидентификация становится важна.

13.6. Сопоставление по шаблону с помощью receive

Большинство полезных протоколов взаимодействия сложнее, чем определенный выше. Возможности, которые предоставляет receiveOnly, весьма ограничены. Например, с помощью receiveOnly довольно сложно реализовать такой маневр, как «получить int или string».

Гораздо более мощным примитивом является функция receive, которая сопоставляет и диспетчирует сообщения в зависимости от их типа. Типичный вызов receive выглядит так:

```
receive(
   (string s) { writeln("Получена строка со значением    s); },
   (int x) { writeln("Получено число со значением "  x); }
);
```

При сопоставлении этого вызова со следующими вызовами send во всех случаях будет наблюдаться совпадение:

```
send(tid, "здравствуй");
send(tid, 5);
send(tid, 'a');
send(tid, 42u);
```

Первый вызов send соответствует типу string и направляется в литерал функции, определенный в receive первым; остальные три вызова соответствуют типу int и передаются во второй функциональный литерал. Кстати, в качестве функций-обработчиков необязательно использовать литералы – какие-то (или даже все) обработчики могут быть адресами именованных функций:

```
void handleString(string s) {     }
receive(
   &handleString,
   (int x) { writeln("Получено число со значением "  x); }
);
```

Сопоставление не является досконально точным; вместо того чтобы требовать точного совпадения, соблюдают обычные правила перегрузки, в соответствии с которыми char и uint могут быть неявно преобразованы в int. При сопоставлении следующих вызовов соответствие, напротив, обнаружено *не будет*:

```
send(tid, "hello"w); // Строка в кодировке UTF-16 (см. раздел 4.5)
send(tid, 5L);       // long
send(tid, 42.0);     // double
```

Когда функция receive видит сообщение неожиданного типа, она не порождает исключение (как это делает receiveOnly). Подсистема обмена сообщениями просто сохраняет неподходящие сообщения в очереди, в народе называемой *почтовым ящиком* (*mailbox*) потока. receive терпеливо ждет, когда в почтовом ящике появится сообщение нужного типа. Такая политика делает receive и протоколы, реализованные на базе этой функции, более гибкими, но и более восприимчивыми к блокированию и переполнению ящика. Одно недоразумение при обмене информацией – и в ящике потока начнут накапливаться сообщения не тех типов, а receive тем временем будет ждать сообщения, которое никогда не придет.

Пользуясь посредническими услугами Tuple, дуэт send/receive с легкостью обрабатывает и группы аргументов. Например:

```
receive(
   (long x, double y) {     },
   (int x) {     }
);
```

соответствуют те же сообщения, что и

```
receive(
   (Tuple!(long, double) tp) {     },
   (int x) { ..  }
);
```

Такой вызов, как send(tid, 5, 6.3), соответствует первому функциональному литералу как первого, так и второго предыдущих примеров.

Существует особая версия receive – функция receiveTimeout, позволяющая потоку предпринять экстренные меры в случае задержки сообщений. У receiveTimeout есть «срок годности»: она завершает свое выполнение по истечении указанного промежутка времени. Об истечении «отпущенного времени» receiveTimeout сообщает, возвращая false:

```
auto gotMessage = receiveTimeout(
   1000, // Время в милисекундах
   (string s) { writeln("Получена строка со значением "  s); },
   (int x) { writeln("Получено число со значением "  x); }
);
if (!gotMessage) {
   stderr.writeln("Выполнение прервано по прошествии одной секунды.");
}
```

13.6.1. Первое совпадение

Рассмотрим пример:

```
receive(
   (long x) {     },
   (string x) { ..  },
   (int x) {     }
);
```

Такой вызов не скомпилируется: receive отвергает этот вызов, поскольку третий обработчик недостижим при любых условиях. Любое отправленное по каналу передачи значение типа int застревает в первом обработчике.

Порядок аргументов receive определяет, каким образом осуществляются попытки сопоставления. Принцип тот же, что и при вычислении блоков catch в инструкции try, но не при объектно-ориентированной диспетчеризации функций. Единого мнения насчет относительных преимуществ и недостатков использования первого совпадения или же лучшего совпадения – нет; достаточно сказать, что, по всей видимости, первое совпадение хорошо подходит для этого конкретного случая receive.

Выполнение принципа первого совпадения обеспечивается функцией receive с помощью простого анализа, выполняемого во время компиляции. Для любых типов сообщения ‹*Сбщ*$_1$› и ‹*Сбщ*$_2$› справедливо, что, если в вызове receive обработчик ‹*Сбщ*$_2$› следует после обработчика ‹*Сбщ*$_1$›, receive гарантирует, что тип ‹*Сбщ*$_2$› *невозможно* неявно преобразовать в тип ‹*Сбщ*$_1$›. Если можно, то это означает, что обработчик ‹*Сбщ*$_1$› будет ловить сообщения ‹*Сбщ*$_2$›, так что в компиляции такому вызову будет отказано. Выполнение этой проверки для предыдущего примера завершается неудачей в процессе той итерации, когда ‹*Сбщ*$_1$› присваивается значение long, а ‹*Сбщ*$_2$› – int.

13.6.2. Соответствие любому сообщению

Что если бы вы пожелали обеспечить просмотр абсолютно всех сообщений в почтовом ящике – например, для уверенности в том, что он не переполнится мусором?

Ответ прост – нужно всего лишь включить обработчик сообщений типа `Variant` последним в список аргументов `receive`. Например:

```
receive(
   (long x) { ... },
   (string x) {     },
   (double x, double y) {     },

   (Variant any) {     }
);
```

Тип `Variant`, определенный в модуле `std.variant`, – это динамический тип, вмещающий ровно одно значение любого другого типа. `receive` воспринимает `Variant` как обобщенный контейнер для любого типа сообщения, а потому вызов `receive` с обработчиком для типа `Variant` всегда будет отработан, если в очереди есть хотя бы одно сообщение.

Расположить обработчик `Variant` в конце цепочки обработки сообщений – хороший способ избавить ваш почтовый ящик от случайных сообщений.

13.7. Копирование файлов – с выкрутасом

Напишем коротенькую программу для копирования файлов – один из популярных способов познакомиться с интерфейсом языка файловой системы. Классический пример в стиле Кернигана и Ричи целиком на паре команд `getchar/putchar` [34, глава 1, с. 15]. Конечно же, чтобы ускорить передачу, «родные» программы системы, копирующие файлы, практикуют буферное чтение и буферную запись, а также используют множество других методов оптимизации, так что написать конкурентоспособную программу было бы сложно, однако параллельные вычисления нам помогут.

Обычный способ копирования файлов:

1. Прочесть данные из исходного файла и поместить в буфер.
2. Если ничего не было прочитано, копирование завершено.
3. Записать данные из буфера в целевой файл.
4. Повторить заново, начиная с шага 1.

Добавление соответствующей обработки ошибок завершит полезную (но не оригинальную) программу. Если размер буфера будет выбран достаточно большим, а оба файла (и источник, и целевой файл) окажутся на одном и том же диске, быстродействие этого алгоритма приблизится к оптимальному.

В наше время файловыми хранилищами могут быть многие физические устройства: жесткие диски, флеш-диски, оптические диски, подсоединенные смартфоны, а также сетевые сервисы удаленного доступа. Эти устройства характеризуются разнообразными показателями задержки и скорости и подключаются с помощью разных аппаратных и программных интерфейсов. Такие интерфейсы могут работать параллельно (а не по одному в каждый момент времени, как предписывает алгоритм в стиле «прочесть данные из буфера/записать данные в буфер»), и именно так и нужно их использовать. В идеале должна поддерживаться максимальная занятость как устройства-источника, так и устройства-получателя, что мы можем изобразить как два потока, работающих по протоколу «поставщик/потребитель»:

1. Породить один дочерний поток, который в цикле ждет сообщений, содержащих буферы памяти, и записывает их в целевой файл.
2. Прочесть данные из исходного файла и разместить их в заново созданном буфере.
3. Если ничего не было прочитано, копирование завершено.
4. Отправить дочернему потоку сообщение, содержащее буфер с прочитанными данными.
5. Повторить, начав с шага 2.

С таким подходом один поток будет работать с источником, а другой – с приемником. В зависимости от природы «исходного пункта» и «пункта назначения» можно получить значительное ускорение. Если скорости устройств сравнимы и невелики относительно пропускной способности шины памяти, теоретически скорость копирования может быть удвоена. Напишем простую программу, которая реализует модель «поставщик/потребитель» и копирует содержимое стандартного потока ввода в стандартный поток вывода:

```
import std.concurrency, std.stdio;

void main() {
   enum bufferSize = 1024 * 100;
   auto tid = spawn(&fileWriter);
   // Цикл чтения
   foreach (ubyte[] buffer; stdin.byChunk(bufferSize)) {
      send(tid, buffer.idup);
   }
}

void fileWriter() {
   // Цикл записи
   for (;;) {
      auto buffer = receiveOnly!(immutable(ubyte)[])();
      stdout.rawWrite(buffer);
   }
}
```

В этой программе данные из основного потока передаются в дочерний поток посредством разделения неизменяемых данных: передаваемые сообщения имеют тип immutable(ubyte)[], то есть являются массивами неизменяемых значений типа ubyte. Эти буферы создаются в цикле foreach при чтении данных из входного потока порциями, каждая из которых имеет тип immutable(ubyte)[] и размер bufferSize. На каждом проходе цикла функция byChunk читает данные во временный буфер (переменную buffer), неизменная копия которого создается свойством idup. Большую часть тяжелой работы выполняет управляющая часть foreach; на долю тела этой конструкции остается лишь создание копии и отправка буфера дочернему потоку. Как уже говорилось, передача данных между потоками возможна благодаря присутствию квалификатора immutable; если заменить idup на dup, вызов send не скомпилируется.

13.8. Останов потока

В приводившихся до сих пор примерах есть кое-что необычное, в частности в функции writer, определенной в разделе 13.5, и в только что определенной функции fileWriter из раздела 13.7: обе функции содержат бесконечный цикл. На самом деле, повнимательнее взглянув на пример с копированием файлов, можно заметить, что main и fileWriter прекрасно понимают друг друга в разговоре о копировании, но никогда не обсуждают друг с другом останов приложения; другими словами, main никогда не говорит fileWriter: «Дело сделано, собирайся и пойдем домой».

Останов многопоточных приложений всегда был делом мудреным. Поток легко запустить, но запустив, трудно остановить; завершение работы приложения – событие асинхронное и может застать приложение за выполнением совершенно произвольной операции. Низкоуровневые API для работы с потоками предоставляют средство для принудительного останова потоков, неизменно сопровождая его предупреждением о чрезмерной грубости этого инструмента и рекомендацией об использовании какого-нибудь более высокоуровневого протокола завершения работы.

D предоставляет простой и надежный протокол останова потоков. Каждый поток обладает *потоком-владельцем*; по умолчанию владельцем считается поток, инициировавший вызов функции spawn. Владельца текущего потока можно изменить динамически, сделав вызов вида setOwner(tid). У каждого потока только один владелец, но сам он может быть владельцем множества потоков.

Самое важное проявление отношения «владелец/собственность» заключается в том, что по завершении выполнения потока-владельца вызовы функции receive в дочернем потоке начнут порождать исключения типа OwnerTerminated. Исключение порождается, только если в очереди к receive больше нет подходящих сообщений и необходимо ждать прихода новых; пока у receive есть что извлечь из ящика, она не породит исключение OwnerTerminated. Другими словами, при останове потока-

владельца вызовы receive (или receiveOnly, коли на то пошло) в дочерних потоках породят исключения тогда и только тогда, когда в противном случае они заблокируют выполнение программы, так как продолжат ожидать сообщение, которое никогда не придет. Отношение владения необязательно однонаправленно. В действительности, возможна ситуация, когда два потока являются владельцами друг друга; в таком случае, какой бы поток ни завершался первым, он оповестит другой поток.

Окинем программу копирования файлов свежим взглядом — с учетом знания об отношении владения. В любой заданный момент времени в полете между основным и второстепенным потоками находится масса сообщений. Чем быстрее выполняются операции чтения по сравнению с операциями записи, тем больше буферов будет находиться в почтовом ящике записывающего потока в ожидании обработки. Возврат из main заставит receive породить исключение, но не раньше, чем будут обработаны ожидающие сообщения. Сразу же после того, как ящик записывающего потока опустеет (а последняя порция данных будет записана в целевой файл), очередной вызов receive породит исключение. Записывающий поток прекращает выполнение по исключению OwnerTerminated; система времени исполнения в курсе, что это за исключение, и просто его игнорирует. Операционная система закрывает стандартные потоки ввода и вывода так, как обычно, и операция копирования успешно завершается.

Может показаться, что в промежутке между моментом отправки последнего сообщения из main и моментом возврата из main (что заставляет receive породить исключение) возникает гонка. Что если исключение «опередит» последнее сообщение — или, хуже того, несколько последних сообщений? На самом деле никакой гонки нет. Поток, отправляющий сообщения, всегда думает о последствиях: последнее сообщение помещается в конец очереди дочернего потока *до* того, как исключение OwnerTerminated начнет свой путь (фактически распространение исключения организуется при помощи той же очереди, что и в случае обычных сообщений). Однако гонка *присутствовала бы*, если бы функция main завершала свое выполнение в тот самый момент, когда другой, третий поток отправлял бы сообщения в очередь fileWriter.

Подобная же цепочка рассуждений показывает, что наш предыдущий простой пример, в котором два потока «в ногу» записывают 200 сообщений, также корректен: функция main завершает свое выполнение после отправки (ждет до конца) последнего сообщения дочернему потоку. Дочерний поток сначала опустошает очередь, а затем заканчивает работу по исключению OwnerTerminated.

Если вы считаете, что для механизма, обрабатывающего завершение выполнения потока, порождение исключения — выбор слишком суровый, то помните, что никто не лишал вас возможности обработать OwnerTerminated явно:

```
// Завершается без исключения
void fileWriter() {
```

```
    // Цикл записи
    for (bool running = true; running; ) {
        receive(
            (immutable(ubyte)[] buffer) { tgt.write(buffer); },
            (OwnerTerminated) { running = false; }
        );
    }
    stderr.writeln("Выполнение завершено без приключений.");
}
```

В данном случае по завершении выполнения main поток fileWriter мирно возвращает управление, и все счастливы. Но что произойдет, если исключение породит дочерний, записывающий поток? Если возникнут проблемы с записью данных в tgt, вызов функции write может завершиться неудачей. В таком случае вызов send из основного потока также завершится неудачей (а именно будет порождено исключение типа OwnerFailed), то есть произойдет как раз то, что ожидалось. Кстати, если дочерний поток завершит свое выполнение обычным способом (а не по исключению), последующие вызовы send, отправлявшие сообщения этому потоку, также завершатся неудачей, но с другим типом исключения – OwnedTerminated.

Рассмотренная программа для копирования файлов более отказоустойчива, чем можно предположить, судя по ее простоте. Тем не менее нужно сказать, что протокол завершения выполнения гладко срабатывает лишь тогда, когда отношения между потоками просты и предельно понятны, и полагаться на него стоит исключительно в таких случаях. А когда в деле замешаны несколько потоков и отношения владения между ними отражаются сложным графом, лучше всего организовать взаимодействие всех этих потоков по протоколам, предусматривающим явное уведомление об окончании обмена данными. В случае примера с копированием файлов можно реализовать следующую простую идею: установить соглашение, по которому отправка буфера нулевого размера записывающему потоку будет означать удачное завершение работы читающим потоком. Получив такое сообщение и завершив запись, записывающий поток также уведомляет поток, осуществлявший чтение, о своем завершении. После чего «читатель», наконец, тоже может завершить свое выполнение. Такой протокол явного уведомления хорошо масштабируется до случаев, когда по пути от «читателя» к «писателю» данные обрабатываются множеством других потоков.

13.9. Передача нештатных сообщений

Допустим, с помощью предположительно прыткой программы, которую мы только что написали, вы копируете большой файл из быстрого локального хранилища на медленный сетевой диск. На полпути возникает ошибка чтения – файл поврежден. Это заставляет read, а затем и main породить исключения, и все происходит тогда, когда множество

буферов находятся в полете, но еще не записаны. Более абстрактно, мы видели, что если поток-владелец завершит свое выполнение *обычным способом*, любой блокирующий вызов receive из принадлежащих ему потоков породит исключение. Но что произойдет, если владелец завершит выполнение по исключению?

Если поток завершается посредством порождения исключения, это знак серьезной проблемы, о которой с должной настойчивостью нужно уведомить дочерние потоки. И, конечно, это выполняется с помощью *нештатного* сообщения.

Вспомните, что функция receive заботится лишь о сообщениях, совпавших с заданными шаблонами, а остальным позволяет накапливаться в очереди. Есть способ внести в это поведение поправку. Поток-отправитель может инициировать обработку сообщения потоком-получателем, вызвав функцию prioritySend вместо send. Эти две функции принимают одни и те же параметры, но ведут себя по-разному, что в действительности отражается на поведении получателя. Передача сообщения типа T с помощью prioritySend заставляет receive в потоке-получателе действовать следующим образом:

- Если вызов receive предусматривает обработку типа T, то сообщение с приоритетом будет извлечено сразу же после завершения обработки текущего сообщения – даже если сообщение с приоритетом пришло позже других обычных (неприоритетных) сообщений. Сообщения с приоритетом всегда помещаются в начало очереди, так что последнее пришедшее сообщение с приоритетом всегда извлекается функцией receive первым (даже если другие сообщения с приоритетом уже ждут).
- Если вызов receive не обрабатывает тип T (то есть совокупность указанных обстоятельств предписывает receive оставить сообщение такого типа в почтовом ящике в ожидании) и T является наследником Exception, то receive напрямую порождает извлеченное сообщение-исключение.
- Если вызов receive не обрабатывает тип T и T не является наследником Exception, то receive порождает исключение типа PriorityMessageException!T. Объект этого исключения содержит копию полученного сообщения в виде внутреннего элемента message.

Если поток завершается по исключению, исключение OwnerFailed распространяется на все потоки, которыми он владеет, с помощью вызова prioritySend. В программе копирования файлов порождение исключения внутри main вызывает порождение исключения и внутри fileWriter (как только там будет вызвана функция receive); в результате, напечатав сообщение об ошибке и вернув ненулевой код выхода, останавливается весь процесс. В отличие от случая с «нормальным» завершением исполнения, в данной ситуации вполне допустимо, что в подвешенном состоянии останутся буферы, которые были уже прочитаны, но еще не записаны.

13.10. Переполнение почтового ящика

Программа для копирования файлов на основе протокола «поставщик/потребитель» работает достаточно хорошо, однако обладает одним важным недостатком. Рассмотрим копирование большого файла, при котором данные передаются между устройствами, скорость доступа к которым существенно различается, например копирование приобретенного законным способом файла с фильмом с внутреннего диска (быстрый доступ) на сетевой диск (вероятно, значительно более медленный доступ). В этом случае поставщик (основной поток, выполняющий функцию `main`) порождает буферы со значительной скоростью, гораздо более высокой, чем скорость, с которой потребитель в состоянии записать их в целевой файл. Разница в скоростях вызывает скопление данных, напрасно занимающих память, которую программа не может использовать для повышения производительности.

Во избежание переполнения почтового ящика, API для параллельных вычислений позволяет задать максимальный размер очереди сообщений, а также действие, предпринимаемое при достижении этого предела. Соответствующие сигнатуры выглядят так:

```
// Внутри std.concurrency
void setMaxMailboxSize(Tid tid, size_t messages,
   bool function(Tid) onCrowdingDoThis);
```

Вызывая `setMailboxSize`, вы устанавливаете для подсистемы параллельных вычислений правило: всякий раз когда требуется отправить новое сообщение, а очередь уже содержит число сообщений, указанное в `messages`, вызывать `onCrowdingDoThis(tid)`. Если `onCrowdingDoThis(tid)` возвращает `false` или порождает исключение, новое сообщение игнорируется. В противном случае еще раз проверяется размер очереди потока, и если выясняется, что он уже меньше, чем размер `messages`, новое сообщение доставляется потоку с идентификатором `tid`. В противном случае весь цикл возобновляется.

Вызов `setMaxMailboxSize` выполняется в потоке, осуществляющем вызов, а не в потоке, этот вызов принимающем. Иными словами, поток, инициирующий отправку сообщения, также является ответственным и за принятие экстренных мер при переполнении почтового ящика получателя. Кажется логичным спросить: почему нельзя расположить этот вызов в потоке-получателе? При расширении масштаба, а именно применительно к программам с большим количеством потоков, такой подход породил бы порочные последствия: потоки, пытающиеся отправить сообщения, угрожали бы лишить трудоспособности потоки с полными ящиками.

Есть ряд предопределенных действий, предпринимаемых в случае, если почтовый ящик полон: заблокировать отправителя до тех пор, пока очередь не станет меньше, породить исключение или проигнорировать новое сообщение. Такие предопределенные действия удобно упакованы:

```
// Внутри std.concurrency
enum OnCrowding { block, throwException, ignore }
void setMaxMailboxSize(Tid tid, size_t messages, OnCrowding doThis);
```

В нашем случае лучше всего попросту блокировать поток-читатель, как только ящик становится слишком большим. Добиться этого можно, вставив вызов

```
setMaxMailboxSize(tid, 1024, OnCrowding.block);
```

сразу же после вызова spawn.

В следующих разделах описываются подходы к организации межпоточной передачи данных, служащие или альтернативой, или дополнением к обмену сообщениями. Обмен сообщениями – рекомендуемый метод организации межпоточного взаимодействия; этот метод легок для понимания, порождает удобный для чтения код, является надежным и масштабируемым. К более низкоуровневым механизмам стоит обращаться лишь в совершенно особых обстоятельствах – и не забывайте, что «особые» обстоятельства не всегда настолько особые, какими кажутся.

13.11. Квалификатор типа shared

Мы уже познакомились с квалификатором shared в разделе 13.3. Для системы типов ключевое слово shared служит сигналом о том, что несколько потоков обладают доступом к одному фрагменту данных. Компилятор тоже признает этот факт и соответственно реагирует, накладывая ограничения на операции с разделяемыми данными, а также посредством генерации особого кода для разрешенных операций.

С помощью глобального определения

```
shared uint threadsCount;
```

в программу на D вводится значение типа shared(uint), что соответствует глобально определенному целому числу без знака в программе на C. Такая переменная видима всем потокам в системе. Примечание в виде shared здорово помогает компилятору: язык «знает», что threadsCount открыт для свободного доступа множеству потоков, и запрещает обращения к этой переменной наивными способами. Например:

```
void bumpThreadsCount() {
   ++threadsCount; // Ошибка!
   // Увеличить на единицу значение типа shared int невозможно!
}
```

Что происходит? Где-то внизу, на машинном уровне, ++threadCount не является атомарной операцией; это сложная операция, представляющая собой последовательность трех простых: прочесть – изменить – записать. Сначала threadCount загружается в регистр, затем значение регистра увеличивается на единицу и, наконец, threadCount записывается обратно в память. Для обеспечения корректности всей сложной операции

эти три шага необходимо выполнять единым блоком. Корректный способ увеличить на единицу разделяемое целое число – воспользоваться одним из специализированных атомарных примитивов из модуля std.concurrency:

```
import std.concurrency;
shared uint threadsCount;

void bumpThreadsCount() {
   // std.concurrency определяет
   // atomicOp(string op)(ref shared uint, int)
   atomicOp!"+="(threadsCount, 1); // Все в порядке
}
```

Поскольку все разделяемые данные тщательно учитываются и находятся под эгидой языка, передавать данные с квалификатором shared разрешается с помощью функций send и receive.

13.11.1. Сюжет усложняется: квалификатор shared транзитивен

В главе 8 объясняется, почему квалификаторы const и immutable должны быть *транзитивными* (свойство, также известное как глубина или рекурсивность): каким бы косвенным путем вы ни следовали, рассматривая «внутренности» неизменяемого объекта, сами данные должны оставаться неизменяемыми. В противном случае гарантии, предоставляемые квалификатором immutable, имели бы силу комментария в коде. Нельзя сказать, что нечто «до определенного момента» неизменяемо (immutable), а дальше меняется. Зато можно говорить, что данные *изменяемы* до определенного момента, а затем становятся совершенно неизменяемыми, вплоть до самых глубоко вложенных элементов. Применив квалификатор immutable, вы сворачиваете на улицу с односторонним движением. Мы уже видели, что присутствие квалификатора immutable облегчает реализацию многих оправдавших себя идиом, не претендующих на свободу программиста, включая функциональный стиль и разделение данных между потоками. Если бы неизменяемость применялась «до определенного момента», то же самое относилось бы и к корректности программы.

Точно такой же ход рассуждений применим и для квалификатора shared. На самом деле, в случае с shared необходимость транзитивности абсолютно очевидна. Приведем пример. Выражение

```
shared int* pInt;
```

в соответствии с синтаксисом квалификаторов (см. раздел 8.2) эквивалентно выражению

```
shared(int*) pInt;
```

Верная интерпретация pInt такова: «Указатель является разделяемым, и данные, на которые он указывает, также разделяемы». При поверх-

ностном, нетранзитивном подходе к разделению pInt превратился бы в «разделяемый указатель на неразделяемую память», и все бы ничего, если бы такой тип данных имел хоть какой-то смысл. Это все равно что сказать: «Я делюсь этим бумажником со всеми; только, пожалуйста, не забывайте, что деньгами из него я делиться не собирался»[1]. Заявление, что потоки разделяют указатель, но не данные, на которые он указывает, возвращает нас к чудесной парадигме программирования на основе системы доверия, которая всегда успешно проваливалась. И причина большей части проблем не чьи-то происки, а честные ошибки. Программное обеспечение имеет большой объем, сложно устроено и постоянно изменяется, что плохо сочетается с обеспечением гарантий на основе соглашений.

Тем не менее есть совершенно логичное понятие «неразделяемый указатель на разделяемые данные». Некоторый поток обладает «личным» указателем, а этот указатель «смотрит» на разделяемые данные. Эту идею легко выразить синтаксически:

```
shared(int)* pInt;
```

Между нами, если бы существовала премия «За лучшее отображение содержания», нотация квалификатор(тип) ее бы отхватила. Эта форма записи совершенна. Синтаксис просто не позволит создать неправильный указатель. Некорректное сочетание синтаксических единиц выглядит так:

```
int shared(*) pInt;
```

Такое выражение не имеет смысла даже синтаксически, поскольку (*) – это не тип (ну да, *на самом деле* этот милый смайлик символизирует циклопа).

Транзитивность квалификатора shared действует не только в отношении указателей, но и в отношении полей объектов-структур и классов: поля разделяемого объекта также автоматически воспринимаются как помеченные квалификатором shared. Подробный разбор порядка взаимодействия этого квалификатора с классами и структурами представлен далее в этой главе.

13.12. Операции с разделяемыми данными и их применение

Работа с разделяемыми данными необычна, поскольку множество потоков могут читать и записывать разделяемые данные в любой момент. Поэтому компилятор заботится о соблюдении целостности данных и причинности всеми операциями с разделяемыми данными.

---

[1] Кстати, воспользовавшись квалификатором const, вы сможете делиться бумажником, зная при этом, что деньги в нем защищены от воров. Стоит лишь ввести тип shared(const(Money)*).

Операции чтения и записи разделяемых (shared) значений разрешены, и гарантированно будут атомарными для следующих типов: числовые типы (кроме real), указатели, массивы, указатели на функции, делегаты и ссылки на классы. Структуру с единственным полем одного из перечисленных типов также можно читать и записывать как неделимый объект. Подчеркнутое отсутствие в списке «разрешенных типов» типа real обусловлено тем, что это единственный тип, зависящий от платформы. Вот почему в плане атомарного разделения компилятор смотрит на real с опаской. На машинах Intel real занимает 80 бит, из-за чего переменным этого типа сложно делать атомарные присваивания в 32-разрядных программах. В любом случае, тип real предназначен для хранения временных результатов высокой точности, а не для обмена данными, так что вряд ли у кого-то возникнет желание разделять значения этого типа.

Для всех числовых типов и указателей на функции справедливо, что значения этих типов с квалификатором shared могут неявно преобразовываться в значения без квалификатора и обратно. Преобразования указателей между shared(T*) и shared(T)* разрешены в обоих направлениях. Арифметические операции на разделяемых числовых типах позволяют выполнять примитивы из модуля std.concurrency.

13.12.1. Последовательная целостность разделяемых данных

Что касается видимости операций над разделяемыми данными между потоками, D предоставляет следующие гарантии:

- порядок выполнения операций чтения и записи разделяемых данных в рамках одного потока соответствует порядку, определенному в исходном коде;
- глобальный порядок выполнения операций чтения и записи разделяемых данных представляет собой некоторое чередование операций чтения и записи, выполнение которых инициируется из разных потоков.

Выбор этих инвариантов кажется вполне резонным, даже очевидным. И на самом деле такие гарантии довольно хорошо гармонируют с моделью вытесняющей многозадачности, реализованной на однопроцессорных системах.

Тем не менее в контексте мультипроцессорных систем такие гарантии слишком строги. Проблема в следующем: для обеспечения этих гарантий необходимо сделать так, чтобы результат выполнения любой операции записи был сразу же виден всем потокам. Единственный способ добиться этого — окружить обращения к разделяемым данным особыми машинными инструкциями (их называют *барьеры памяти*), которые обеспечивали бы соответствие порядка, в котором выполняются операции чтения и записи разделяемых данных, порядку обновления этих

данных в глазах всех запущенных потоков. Присутствие замысловатых иерархий кэшей значительно удорожает такую сериализацию. Кроме того, непоколебимая приверженность принципу последовательной целостности заставляет отказаться от переупорядочивания операций – основы множества способов оптимизации на уровне компилятора. В сочетании друг с другом эти два ограничения ведут к резкому замедлению – вплоть до одного порядка единиц измерения.

Хорошая новость заключается в том, что такая потеря скорости имеет место лишь в отношении разделяемых данных, которые используются достаточно редко. В реальных ситуациях большинство данных не разделяются, а потому нет необходимости, чтобы в их отношении соблюдался принцип последовательной целостности. Компилятор оптимизирует код, используя неразделяемые данные на всю катушку, в полной уверенности, что другой поток никогда к ним не обратится, и относится с осторожностью лишь к разделяемым данным. Повсеместно используемый и рекомендуемый прием для работы с разделяемыми данными – копировать значения разделяемых переменных в локальные рабочие копии потоков, работать с копиями и затем присваивать копии тем же разделяемым переменным.

13.13. Синхронизация на основе блокировок через синхронизированные классы

Традиционно популярный метод многопоточного программирования – *синхронизация на основе блокировок*. В соответствии с этим подходом разделяемые данные защищаются с помощью мьютексов – объектов синхронизации, обеспечивающих переход от параллельного к последовательному исполнению фрагментов кода, которые или временно нарушают когерентность данных, или могут видеть эти временные нарушения. Такие фрагменты кода называют *критическими участками*[1].

Корректность программы, основанной на блокировках, обеспечивается за счет ввода упорядоченного, последовательного доступа к разделяемым данным. Поток, которому требуется обратиться к фрагменту разделяемых данных, должен захватить (заблокировать) мьютекс, обработать данные, а затем освободить (разблокировать) мьютекс. В любой заданный момент времени мьютексом может обладать только один поток, благодаря чему и обеспечивается переход к последовательному выполнению: если захватить один и тот же мьютекс желают несколько потоков, то «выигрывает» лишь один, а остальные скромно ожидают своей

[1] Возможна путаница из-за того, что Windows использует термин «критический участок» для обозначения легковесных объектов мьютексов, защищающих критические участки, а «мьютекс» – для более массивных мьютексов, с помощью которых организуется передача данных между процессами.

очереди. (Способ обслуживания очереди, то есть порядок очередности, играет важную роль и может довольно заметно сказываться на работе приложений и операционной системы.)

По всей вероятности, «Здравствуй, мир!» многопоточного программирования — это пример с банковским счетом: объект, доступный множеству потоков, должен предоставить безопасный интерфейс для пополнения счета и извлечения денежных средств со счета. Вот однопоточная, базовая версия программы, позволяющей выполнять эти действия:

```
import std.contracts;

// Однопоточный банковский счет
class BankAccount {
   private double _balance;
   void deposit(double amount) {
      _balance += amount;
   }
   void withdraw(double amount) {
      enforce(_balance >= amount);
      _balance -= amount;
   }
   @property double balance() {
      return _balance;
   }
}
```

В отсутствие потоков операции += и -= слегка вводят в заблуждение: они «выглядят» как атомарные, но таковыми не являются — обе операции состоят из тройки простых операций «прочесть — изменить — записать». На самом деле, выражение `_balance += amount` кодируется как `_balance = _balance + amount`, а значит, процессор загружает `_balance` и `_amount` в собственную оперативную память (регистры или внутренний стек), складывает их, а затем переводит результат обратно в `_balance`.

Незащищенные параллельные операции типа «прочесть — изменить — записать» становятся причиной некорректного поведения программы. Скажем, баланс вашего счета характеризует истинное выражение `_balance == 100.0`. Некоторый поток, запуск которого был спровоцирован требованием зачислить денежные средства по чеку, делает вызов `deposit(50)`. Сразу же после загрузки из памяти значения `100.0` выполнение этой операции прерывает другой поток, осуществляющий вызов `withdraw(2.5)`. (Это вы в кофейне на углу оплачиваете латте своей дебетовой картой.) Пусть ничто не вклинивается в обработку этого вызова, так что поток, запущенный из кофейни, удачно обновляет поле `_balance`, и оно принимает значение `97.5`. Однако это событие происходит совершенно без ведома депонирующего потока, который уже загрузил число `100` в регистр ЦПУ и все еще считает, что это верное количество. При вычислении нового значения баланса вызов `deposit(50)` получает `150` и записывает это число назад в переменную `_balance`. Это типичное *состояние*

гонки. Поздравляю, вы получили бесплатный кофе (но остерегайтесь: книжные примеры с ошибками еще могут работать на вас, а готовый код с ошибками – нет). Для организации корректной синхронизации многие языки предоставляют специальное средство – тип Mutex, который используется в программах, работающих с несколькими потоками на основе блокировок, для защиты доступа к balance:

```
// Этот код написан не на D
// Многопоточный банковский счет на языке с явным обращением к мьютексам
class BankAccount {
   private double _balance;
   private Mutex _guard;
   void deposit(double amount) {
      _guard.lock();
      _balance += amount;
      _guard.unlock();
   }
   void withdraw(double amount) {
      _guard.lock();
      try {
         enforce(_balance >= amount);
        _balance -= amount;
      } finally {
        _guard.unlock();
      }
   }
   @property double balance() {
     _guard.lock();
      double result = _balance;
     _guard.unlock();
      return result;
   }
}
```

Все операции над _balance теперь защищены, поскольку для доступа к этому полю необходимо заполучить _guard. Может показаться, что приставлять к _balance охранника в виде _guard излишне, так как значения типа double можно читать и записывать «в один присест», однако защита должна здесь присутствовать по причинам, скрытым многочисленными завесами майи. Вкратце, из-за сегодняшних агрессивно оптимизирующих компиляторов и нестрогих моделей памяти *любое* обращение к разделяемым данным должно сопровождаться своего рода секретным соглашением между записывающим потоком, читающим потоком и оптимизирующим компилятором; одно неосторожное чтение разделяемых данных – и вы оказываетесь в мире боли (хорошо, что D намеренно запрещает такую «наготу»). Первая и наиболее очевидная причина такого положения дел в том, что оптимизирующий компилятор, не замечая каких-либо попыток синхронизировать доступ к данным с вашей стороны, ощущает себя вправе оптимизировать код с обращениями к _balance, удерживая значение этого поля в регистре. Вторая

причина в том, что во всех случаях, кроме самых тривиальных, компилятор *и* ЦПУ ощущают себя вправе свободно переупорядочивать незащищенные, не снабженные никаким дополнительным описанием обращения к разделяемым данным, поскольку считают, что имеют дело с данными, принадлежащими лично одному потоку. (Почему? Да потому что чаще всего так и бывает, оптимизация порождает код с самым высоким быстродействием, и в конце концов, почему должны страдать плебеи, а не избранные и достойные?) Это один из тех моментов, с помощью которых современная многопоточность выражает свое пренебрежение к интуиции и сбивает с толку программистов, сведущих в классической многопоточности. Короче, чтобы обеспечить заключение секретного соглашения, потребуется обязательно синхронизировать обращения к свойству `_balance`.

Чтобы гарантировать корректное снятие блокировки с `Mutex` в условиях возникновения исключений и преждевременных возвратов управления, языки, в которых продолжительность жизни объектов контекстно ограничена (то есть деструкторы объектов вызываются на выходе из областей видимости этих объектов), определяют вспомогательный тип `Lock`, который устанавливает блок в конструкторе и снимает его в деструкторе. Эта идея развилась в самостоятельную идиому, известную как *контекстное блокирование* [50]. Приложение этой идиомы к классу `BankAccount` выглядит так:

```
// Версия C++: банковский счет, защищенный методом контекстного блокирования
class BankAccount {
private:
   double _balance;
   Mutex _guard;
public:
   void deposit(double amount) {
       Lock lock = Lock(_guard);
      balance += amount;
   }
   void withdraw(double amount) {
       Lock lock = Lock(_guard);
      enforce(_balance >= amount);
      balance -= amount;
   }
   double balance() {
       Lock lock = Lock(_guard);
      return _balance;
   }
}
```

Благодаря введению типа `Lock` код упрощается и повышается его корректность: ведь соблюдение парности операций установления и снятия блока теперь гарантировано, поскольку они выполняются автоматически. Java, C# и другие языки еще сильнее упрощают работу с блокировками, встраивая `_guard` в объекты в качестве скрытого внутреннего

элемента и приподнимая логику блокирования вверх, до уровня сигнатуры метода. Наш пример, реализованный на Java, выглядел бы так:

```
// Версия Java: банковский счет, защищенный методом контекстного
// блокирования, автоматизированного с помощью инструкции synchronized
class BankAccount {
   private double _balance;
   public synchronized void deposit(double amount) {
      _balance += amount;
   }
   public synchronized void withdraw(double amount) {
      enforce(_balance >= amount);
      _balance -= amount;
   }
   public synchronized double balance() {
      return _balance;
   }
}
```

Соответствующий код на C# выглядит так же, за исключением того, что ключевое слово `synchronized` должно быть заменено на `[MethodImpl(MethodImplOptions.Synchronized)]`.

Итак, вы только что узнали хорошую новость: небольшие программы, основанные на блокировках, легки для понимания и, кажется, неплохо работают. Плохая новость в том, что при большом масштабе очень сложно сопоставлять должным образом блокировки и данные, выбирать контекст и «калибр» блокирования и последовательно устанавливать блокировки, затрагивающие сразу несколько объектов (не говоря о том, что последнее может привести к тому, что взаимозаблокированные потоки, ожидая завершения работы друг друга, попадают в *тупик*). В старые добрые времена классического многопоточного программирования подобные проблемы весьма осложняли кодирование на основе блокировок; современная многопоточность (ориентированная на множество процессоров, с нестрогими моделями памяти и дорогим разделением данных) поставила практику программирования с блокировками под удар [53]. Тем не менее синхронизация на основе блокировок все еще полезна для реализации множества задумок.

Для организации синхронизации с помощью блокировок D предоставляет лишь ограниченные средства. Эти границы установлены намеренно: преимущество в том, что таким образом обеспечиваются серьезные гарантии. Что касается случая с `BankAccount`, версия D очень проста:

```
// Версия D: банковский счет, реализованный
// с помощью синхронизированного класса
synchronized class BankAccount {
   private double _balance;
   void deposit(double amount) {
      _balance += amount;
   }
   void withdraw(double amount) {
```

```
        enforce(_balance >= amount);
        _balance -= amount;
    }
    @property double balance() {
        return _balance;
    }
}
```

D поднимает ключевое слово synchronized на один уровень выше, так чтобы оно применялось к целому классу[1]. Благодаря этому маневру класс BankAccount, реализованный на D, получает возможность предоставлять более серьезные гарантии: даже если бы вы пожелали совершить ошибку, при таком синтаксисе нет способа оставить открытой хоть какую-нибудь дверь с черного хода для несинхронизированных обращений к _balance. Если бы в D позволялось смешивать использование синхронизированных и несинхронизированных методов в рамках одного класса, все обещания, данные синхронизированными методами, оказались бы нарушенными. На самом деле опыт с синхронизацией на уровне методов показал, что лучше всего синхронизировать или все методы, или ни один из них; классы двойного назначения приносят больше проблем, чем удобств.

Объявляемый на уровне класса атрибут synchronized действует на объекты типа shared(BankAccount) и автоматически превращает параллельное выполнение вызовов любых методов класса в последовательное. Кроме того, синхронизированные классы характеризуются возросшей строгостью проверок уровня защиты их внутренних элементов. Вспомните, в соответствии с разделом 11.1 обычные проверки уровня защиты в общем случае позволяют обращаться к любым не общедоступным (public) внутренним элементам модуля любому коду внутри этого модуля. Только не в случае синхронизированных классов – классы с атрибутом synchronized подчиняются следующим правилам:

- объявлять общедоступные (public) данные и вовсе запрещено;
- право доступа к защищенным (protected) внутренним элементам есть только у методов текущего класса и его потомков;
- право доступа к закрытым (private) внутренним элементам есть только у методов текущего класса.

13.14. Типизация полей в синхронизированных классах

В соответствии с правилом транзитивности для разделяемых (shared) объектов разделяемый объект класса распространяет квалификатор shared на свои поля. Очевидно, что атрибут synchronized привносит неко-

[1] Впрочем, D разрешает объявлять синхронизированными отдельные методы класса (в том числе статические). – *Прим. науч. ред.*

торый дополнительный закон и порядок, что отражается в нестрогой проверке типов полей внутри методов синхронизированных классов. Ключевое слово synchronized должно предоставлять серьезные гарантии, поэтому его присутствие своеобразно отражается на семантической проверке полей, в чем прослеживается настолько же своеобразная семантика самого атрибута synchronized.

Защита синхронизированных методов от гонок *временна* и *локальна*. Свойство временности означает, что как только метод возвращает управление, поля от гонок больше не защищаются. Свойство локальности подразумевает, что synchronized обеспечивает защиту данных, встроенных непосредственно в объект, но не данных, на которые объект ссылается косвенно (то есть через ссылки на классы, указатели или массивы). Рассмотрим каждое из этих свойств по очереди.

13.14.1. Временная защита == нет утечкам

Возможно, это не вполне очевидно, но «побочным эффектом» временной природы synchronized становится формирование следующего правила: ни один адрес поля не в состоянии «утечь» из синхронизированного кода. Если бы такое произошло, некоторый другой фрагмент кода получил бы право доступа к некоторым данным за пределами временной защиты, даруемой синхронизацией на уровне методов.

Компилятор пресечет любые поползновения возвратить из метода ссылку или указатель на поле или передать значение поля по ссылке или по указателю в некоторую функцию. Покажем, в чем смысл этого правила, на следующем примере:

```
double * nyukNyuk[1]; // Обратите внимание: без shared

void sneaky(ref double r) { nyukNyuk = &r; }

synchronized class BankAccount {
   private double _balance;
   void fun() {
      nyukNyuk = &_balance; // Ошибка! (как и должно быть в этом случае)
      sneaky(_balance);     // Ошибка! (как и должно быть в этом случае)
   }
}
```

В первой строке fun осуществляется попытка получить адрес _balance и присвоить его глобальной переменной. Если бы эта операция завершилась успехом, гарантии системы типов превратились бы в ничто – с момента «утечки» адреса появилась бы возможность обращаться к разделяемым данным через неразделяемое значение. Присваивание не проходит проверку типов. Вторая операция чуть более коварна в том смысле,

[1] nyukNyuk («няк-няк») – «фирменный» смех комика Керли Ховарда. – *Прим. пер.*

что предпринимает попытку создать псевдоним более изощренным способом – через вызов функции, принимающей параметр по ссылке. Такое тоже не проходит; передача значения с помощью ref фактически влечет получение адреса до совершения вызова. Операция получения адреса запрещена, так что и вызов завершается неудачей.

13.14.2. Локальная защита == разделение хвостов

Защита, которую предоставляет synchronized, обладает еще одним важным качеством – она локальна. Имеется в виду, что она не обязательно распространяется на какие-либо данные помимо непосредственных полей объекта. Как только на горизонте появляются косвенности, гарантия того, что потоки будут обращаться к данным по одному, практически утрачивается. Если считать, что данные состоят из «головы» (часть, расположенная в физической памяти, которую занимает объект класса BankAccount) и, возможно, «хвоста» (косвенно доступная память), то можно сказать, что синхронизированный класс в состоянии защитить лишь «голову» данных, в то время как «хвост» остается разделяемым (shared). По этой причине типизация полей синхронизированного (synchronized) класса внутри метода выполняется особым образом:

- значения любых числовых типов не разделяются (у них нет хвоста), так что с ними можно обращаться как обычно (не применяется атрибут shared);
- поля-массивы, тип которых объявлен как T[], получают тип shared(T)[]; то есть голова (границы среза) не разделяется, а хвост (содержимое массива) остается разделяемым;
- поля-указатели, тип которых объявлен как T*, получают тип shared(T)*; то есть голова (сам указатель) не разделяется, а хвост (данные, на которые указывает указатель) остается разделяемым;
- поля-классы, тип которых объявлен как T, получают тип shared(T). К классам можно обратиться лишь по ссылке (это делается автоматически), так что они представляют собой «сплошной хвост».

Эти правила накладываются поверх правила о запрете «утечек», описанного в предыдущем разделе. Прямое следствие такой совокупности правил: операции, затрагивающие непосредственные поля объекта, внутри метода можно свободно переупорядочивать и оптимизировать, как если бы разделение этих полей было временно остановлено – а именно это и делает synchronized.

Иногда один объект полностью владеет другим. Предположим, что класс BankAccount сохраняет все свои предыдущие транзакции в списке значений типа double:

```
// Не синхронизируется и вообще понятия не имеет о потоках
class List(T) {
   ..
   void append(T value) {
```

```
        ...
    }
}

// Ведет список транзакций
synchronized class BankAccount {
    private double _balance;
    private List!double _transactions;
    void deposit(double amount) {
        _balance += amount;
        _transactions.append(amount);
    }
    void withdraw(double amount) {
        enforce(_balance >= amount);
        _balance -= amount;
        _transactions.append(-amount);
    }
    @property double balance() {
        return _balance;
    }
}
```

Класс List не проектировался специально для разделения между потоками, поэтому он не использует никакой механизм синхронизации, но к нему и на самом деле никогда не обращаются параллельно! Все обращения к этому классу замурованы в объекте класса BankAccount и полностью защищены, поскольку находятся внутри синхронизированных методов. Если предполагать, что List не станет затевать никаких безумных проделок вроде сохранения некоторого внутреннего указателя в глобальной переменной, такой код должен быть вполне приемлемым.

К сожалению, это не так. В языке D код, подобный приведенному выше, не заработает никогда, поскольку вызов метода append применительно к объекту типа shared(List!double) некорректен. Одна из очевидных причин отказа компилятора от такого кода в том, что компиляторы никому не верят на слово. Класс List может хорошо себя вести и все такое, но компилятору потребуется более веское доказательство того, что за его спиной не происходит никакое создание псевдонимов разделяемых данных. В теории компилятор мог бы пойти дальше и проверить определение класса List, однако List, в свою очередь, мог бы использовать другие компоненты, расположенные в других модулях, так что не успеете вы сказать «межпроцедурный анализ», как код «на обещаниях» начнет выходить из-под контроля.

Межпроцедурный анализ — это техника, применяемая компиляторами и анализаторами программ для доказательства справедливости предположений о программе с помощью одновременного рассмотрения сразу нескольких функций. Такие алгоритмы анализа обычно обладают низкой скоростью, начинают хуже работать с ростом программы и являются заклятыми врагами раздельной компиляции. Некоторые системы

используют межпроцедурный анализ, но большинство современных языков (включая D) выполняют все проверки типов, не прибегая к этой технике.

Альтернативное решение проблемы с подобъектом-собственностью – ввести новые квалификаторы, которые бы описывали отношения владения, такие как «класс BankAccount является владельцем своего внутреннего элемента _transactions, следовательно, мьютекс BankAccount также обеспечивает последовательное выполнение операций над _transactions». При верном расположении таких примечаний компилятор смог бы получить подтверждение того, что объект _transactions полностью инкапсулирован внутри BankAccount, а потому безопасен в использовании, и к нему можно обращаться, не беспокоясь о неуместном разделении. Системы и языки, работающие по такому принципу [25, 2, 11, 6], уже были представлены, однако в настоящий момент они погоды не делают. Ввод явного указания имеющихся отношений владения свидетельствует о появлении в языке и компиляторе значительных сложностей. Учитывая, что в настоящее время синхронизация на основе блокировок борется за существование, D поостерегся усиливать поддержку этой ущербной техники программирования. Не исключено, что это решение еще будет пересмотрено (для D были предложены системы моделирования отношений владения [42]), но на настоящий момент, чтобы реализовать некоторые проектные решения, основанные на блокировках, приходится, как объясняется далее, переступать границы системы типов.

13.14.3. Принудительные идентичные мьютексы

D позволяет сделать динамически то, что система типов не в состоянии гарантировать статически: отношение «владелец/собственность» в контексте блокирования. Для этого предлагается следующая глобально доступная базовая функция:

```
// Внутри object.d
setSameMutex(shared Object ownee, shared Object owner);
```

Объект obj некоторого класса может сделать вызов obj.setSameMutex(owner)[1], и в результате вместо текущего объекта синхронизации obj начнет использовать тот же объект синхронизации, что и объект owner. Таким способом можно гарантировать, что при блокировке объекта owner блокируется и объект obj. Посмотрим, как это сработает применительно к нашим подопытным классам BankAccount и List.

```
// В курсе о существовании потоков
synchronized class List(T) {
```

[1] На момент выхода книги возможность вызова функций как псевдочленов (см. раздел 5.9) не была реализована полностью, и вместо кода obj.setSameMutex(owner) нужно было писать setSameMutex(obj, owner). Возможно, все уже изменилось. – *Прим. науч. ред.*

```
    void append(T value) {
        ...
    }
}

// Ведет список транзакций
synchronized class BankAccount {
    private double _balance;
    private List!double _transactions;

    this() {
        // Счет владеет списком
        setSameMutex(_transactions, this);
    }

}
```

Необходимое условие работы такой схемы – синхронизация обращений к List (объекту-собственности). Если бы к объекту _transactions применялись лишь обычные правила для полей, впоследствии при выполнении над ним операций он бы просто заблокировался в соответствии с этими правилами. Но на самом деле при обращении к _transactions происходит кое-что необычное: осуществляется явный захват мьютекса объекта типа BankAccount. При такой схеме мы получаем довольный компилятор: он думает, что каждый объект блокируется по отдельности. Довольна и программа: на самом деле единственный мьютекс контролирует как объект типа BankAccount, так и подобъект типа List. Захват мьютекса поля _transactions – это в действительности захват уже заблокированного мьютекса объекта this. К счастью, такой рекурсивный захват уже заблокированного, не запрашиваемого другими потоками мьютекса обходится относительно дешево, так что представленный в примере код корректен и не снижает производительность программы за счет частого блокирования.

13.14.4. Фильм ужасов: приведение от shared

Продолжим работать с предыдущим примером. Если вы абсолютно уверены в том, что желаете возвести список _transactions в ранг святой частной собственности объекта типа BankAccount, то можете избавиться от shared и использовать _transactions без учета потоков:

```
// Не синхронизируется и вообще понятия не имеет о потоках
class List(T) {

    void append(T value) {

    }
}

synchronized class BankAccount {
```

```
    private double _balance;
    private List!double _transactions;
    void deposit(double amount) {
       _balance += amount;
       (cast(List!double) _transactions).append(amount);
    }
    void withdraw(double amount) {
       enforce(_balance >= amount);
       _balance -= amount;
       (cast(List!double) _transactions).append(-amount);
    }
    @property double balance() {
       return _balance;
    }
}
```

На этот раз код с несинхронизированным классом List и компилируется, и запускается. Однако есть одно «но»: теперь корректность основанной на блокировках дисциплины в программе гарантируете вы, а не система типов языка, так что ваше положение не намного лучше, чем при использовании языков с разделением данных по умолчанию. Преимущество, которым вы все же можете наслаждаться, состоит в том, что приведения типов локализованы, а значит, их легко находить, то есть тщательно рассмотреть обращения к cast на предмет ошибок – не проблема.

13.15. Взаимоблокировки и инструкция synchronized

Если пример с банковским счетом – это «Здравствуй, мир!» программ, использующих потоки, то пример с переводом средств со счета на счет, надо полагать, – соответствующее (но более мрачное) введение в проблему межпоточных взаимоблокировок. Условия для задачи с переводом средств формулируются так: пусть даны два объекта типа BankAccount (скажем, checking и savings); требуется определить атомарный перевод некоторого количества денежных средств с одного счета на другой.

Типичное наивное решение выглядит так:

```
// Перевод средств. Версия 1: не атомарная
void transfer(shared BankAccount source, shared BankAccount target,
      double amount) {
   source.withdraw(amount);
   target.deposit(amount);
}
```

Тем не менее эта версия не атомарна; в промежутке между двумя вызовами в теле transfer деньги отсутствуют на обоих счетах. Если точно в этот момент времени другой поток выполнит функцию inspectForAuditing, обстановка может обостриться.

Чтобы сделать операцию перевода средств атомарной, потребуется осуществить захват скрытых мьютексов двух объектов за пределами их методов, в начале функции transfer. Это можно организовать с помощью инструкций synchronized:

```
// Перевод средств. Версия 2: ГЕНЕРАТОР ПРОБЛЕМ
void transfer(shared BankAccount source, shared BankAccount target,
      double amount) {
   synchronized (source) {
      synchronized (target) {
         source.withdraw(amount);
         target.deposit(amount);
      }
   }
}
```

Инструкция synchronized захватывает скрытый мьютекс объекта на время выполнения своего тела. За счет этого вызванные методы данного объекта имеют преимущество уже установленного блока.

Проблема со второй версией функции transfer в том, что она предрасположена к взаимоблокировкам (тупикам): если два потока попытаются выполнить операции перевода между одними и теми же счетами, но *в противоположных направлениях*, то эти потоки могут заблокировать друг друга навсегда. Поток, пытающийся перевести деньги со счета checking на счет savings, блокирует счет checking, а другой поток, пытающийся перевести деньги со счета savings на счет checking, совершенно симметрично умудряется заблокировать счет savings. Этот момент характеризуется тем, что каждый из потоков удерживает свой мьютекс, но для продолжения работы каждому из потоков необходим мьютекс другого потока. К согласию такие потоки не придут никогда.

Решить эту проблему позволяет инструкция synchronized с *двумя* аргументами:

```
// Перевод средств. Версия 3: верная
void transfer(shared BankAccount source, shared BankAccount target, double
amount) {
   synchronized (source, target) {
      source.withdraw(amount);
      target.deposit(amount);
   }
}
```

Синхронизация обращений сразу к нескольким объектам с помощью одной и той же инструкции synchronized и последовательная синхронизация каждого из этих объектов – разные вещи. Сгенерированный код захватывает мьютексы всегда в том же порядке во всех потоках, невзирая на синтаксический порядок, в котором вы укажете объекты синхронизации. Таким образом, взаимоблокировки предотвращаются.

В случае эталонной реализации компилятора истинный порядок установки блокировок соответствует порядку увеличения адресов объектов. Но здесь подходит любой порядок, лишь бы он учитывал все объекты.

Инструкция `synchronized` с несколькими аргументами помогает, но, к сожалению, не всегда. В общем случае действия, вызывающие взаимоблокировку, могут быть «территориально распределены»: один мьютекс захватывается в одной функции, затем другой – в другой и так далее до тех пор, пока круг не замкнется и не возникнет тупик. Однако `synchronized` со множеством аргументов дает дополнительные знания о проблеме и способствует написанию корректного кода с блочным захватом мьютексов.

13.16. Кодирование без блокировок с помощью разделяемых классов

Теория синхронизации, основанной на блокировках, сформировалась в 1960-х. Но уже к 1972 году [23] исследователи стали искать пути исключения из многопоточных программ медленных, неуклюжих мьютексов, насколько это возможно. Например, операции присваивания с некоторыми типами можно было выполнять атомарно, и программисты осознали, что охранять такие присваивания с помощью захвата мьютексов нет нужды. Кроме того, некоторые процессоры стали выполнять транзакционно и более сложные операции, такие как атомарное увеличение на единицу или «проверить-и-установить». Около тридцати лет спустя, в 1990 году, появился луч надежды, который выглядел вполне определенно: казалось, должна отыскаться какая-то хитрая комбинация регистров для чтения и записи, позволяющая избежать тирании блокировок. И в этот момент появилась полная плодотворных идей работа, которая положила конец исследованиям в этом направлении, предложив другое.

Статья Мориса Херлихи «Синхронизация без ожидания» (1991) [3] ознаменовала мощный рывок в развитии параллельных вычислений. До этого разработчикам и аппаратного, и программного обеспечения было одинаково неясно, с какими примитивами синхронизации лучше всего работать. Например, процессор, который поддерживает атомарные операции чтения и записи значений типа `int`, интуитивно могли счесть менее мощным, чем тот, который помимо названных операций поддерживает еще и атомарную операцию `+=`, а третий, который вдобавок предоставляет атомарную операцию `*=`, казался еще мощнее. В общем, чем больше атомарных примитивов в распоряжении пользователя, тем лучше.

Херлихи разгромил эту теорию, в частности показав фактическую бесполезность казавшихся мощными примитивов синхронизации, таких как «проверить-и-установить», «получить-и-сложить» и даже глобальная разделяемая очередь типа FIFO. В свете этих *парадоксов* мгновенно развеялась иллюзия, что из подобных механизмов можно добыть маги-

ческий эликсир для параллельных вычислений. К счастью, помимо получения этих неутешительных результатов Херлихи доказал справедливость *выводов об универсальности*: определенные примитивы синхронизации могут теоретически синхронизировать любое количество параллельно выполняющихся потоков. Поразительно, но реализовать «хорошие» примитивы ничуть не труднее, чем «плохие», причем на невооруженный глаз они не кажутся особенно мощными. Из всех полезных примитивов синхронизации прижился лишь один, известный как сравнение с обменом (compare-and-swap). Сегодня этот примитив реализует фактически любой процессор. Семантика операции сравнения с обменом:

```
// Эта функция выполняется атомарно
bool cas(T)(shared(T) * here, shared(T) ifThis, shared(T) writeThis) {
   if (*here == ifThis) {
      *here = writeThis;
      return true;
   }
   return false;
}
```

В переводе на обычный язык операция cas атомарно сравнивает данные в памяти по заданному адресу с заданным значением и, если значение в памяти равно переданному явно, сохраняет новое значение; в противном случае не делает ничего. Результат операции сообщает, выполнялось ли сохранение. Операция cas целиком атомарна и должна предоставляться в качестве примитива. Множество возможных типов T ограничено целыми числами размером в слово той машины, где будет выполняться код (то есть 32 и 64 бита). Все больше машин предоставляют операцию *сравнения с обменом для аргументов размером в двойное слово* (*double-word compare-and-swap*), иногда ее называют cas2. Операция cas2 автоматически обрабатывает 64-битные данные на 32-разрядных машинах и 128-битные данные на 64-разрядных машинах. Ввиду того что все больше современных машин поддерживают cas2, D предоставляет операцию сравнения с обменом для аргументов размером в двойное слово под тем же именем (cas), под которым фигурирует и перегруженная внутренняя функция. Так что в D можно применять операцию cas к значениям типов int, long, float, double, любых массивов, любых указателей и любых ссылок на классы.

13.16.1. Разделяемые классы

Херлиховские доказательства универсальности спровоцировали появление и рост популярности множества структур данных и алгоритмов в духе нарождающегося «программирования на основе cas». Но есть один нюанс: хотя реализация на основе cas и возможна теоретически для любой задачи синхронизации, но никто не сказал, что это легко. Определение структур данных и алгоритмов на основе cas и особенно доказательство корректности их работы – дело нелегкое. К счастью, однажды

определив и инкапсулировав такую сущность, ее можно повторно использовать для решения самых разных задач [57].

Чтобы ощутить благодать программирования без блокировок на основе cas, воспользуйтесь атрибутом shared применительно к классу или структуре:

```
shared struct LockFreeStruct {

}
shared class LockFreeClass {

}
```

Обычные правила относительно транзитивности в силе: разделяемость распространяется на поля структуры или класса, а методы не предоставляют никакой особой защиты. Все, на что вы можете рассчитывать, – это атомарные присваивания, вызовы cas, уверенность в том, что ни компилятор, ни машина не переупорядочат операции, и собственная безграничная самоуверенность. Однако остерегайтесь: если написание кода – ходьба, а передача сообщений – бег трусцой, то программирование без блокировок – Олимпийские игры, не меньше.

13.16.2. Пара структур без блокировок

Для разминки реализуем стек без блокировок. Основная идея проста: стек моделируется с помощью односвязного списка, операции вставки и удаления выполняются для элементов, расположенных в начале этого списка:

```
shared struct Stack(T) {
   private shared struct Node {
      T _payload;
      Node * _next;
   }
   private Node * _root;

   void push(T value) {
      auto n = new Node(value);
      shared(Node)* oldRoot;
      do {
         oldRoot = _root;
         n._next = oldRoot;
      } while (!cas(&_root, oldRoot, n));
   }

   shared(T)* pop() {
      typeof(return) result;
      shared(Node)* oldRoot;
      do {
         oldRoot = _root;
```

```
            if (!oldRoot) return null;
            result = & oldRoot._payload;
        } while (!cas(&_root, oldRoot, oldRoot._next));
        return result;
    }
}
```

Stack является разделяемой структурой, отсюда прямое следствие: внутри нее практически все тоже разделяется. Внутренний тип Node – классическая структура «полезные данные + указатель», а сам тип Stack хранит указатель на начало списка.

Циклы do/while в теле обеих функций, реализующих базовые операции над стеком, могут показаться странноватыми, но в них нет ничего особенного; медленно, но верно они прокладывают глубокую борозду в коре головного мозга каждого будущего эксперта в cas-программировании. Функция push работает так: сначала создается новый узел, в котором будет сохранено новое значение. Затем в цикле переменной _root присваивается указатель на новый узел, но *только* если тем временем никакой другой поток не изменил ее! Вполне возможно, что другой поток также выполнил какую-то операцию со стеком, так что функции push нужно удостовериться в том, что указатель на начало стека, которому, как предполагается, соответствует значение переменной oldRoot, не изменился за время подготовки нового узла.

Метод pop возвращает результат не по значению, а через указатель. Причина в том, что pop может обнаружить очередь пустой, ведь это не является нештатной ситуацией (как было бы, будь перед нами стек, предназначенный лишь для последовательных вычислений). В случае разделяемого стека проверка наличия элемента, его удаление и возврат составляют одну согласованную операцию. За исключением возвращения результата, функция pop по реализации напоминает функцию push: замена _root выполняется с большой осторожностью, так чтобы никакой другой поток не изменил значение этой переменной, пока извлекаются полезные данные. В конце цикла извлеченное значение отсутствует в стеке и может быть спокойно возвращено инициатору вызова.

Если реализация класса Stack не показалась вам такой уж сложной, возьмемся за реализацию более богатого односвязного интерфейса; в конце концов большая часть инфраструктуры уже выстроена в рамках класса Stack.

К сожалению, в случае со списком все угрожает быть гораздо сложнее. Насколько сложнее? Нечеловечески сложнее. Одна из фундаментальных проблем – вставка и удаление узлов в произвольных позициях списка. Предположим, есть список значений типа int, а в нем есть узел с числом 5, за которым следует узел с числом 10, и требуется удалить узел с числом 5. Тут проблем нет – просто пустите в ход волшебную операцию cas, чтобы нацелить указатель _root на узел с числом 10. Проблема в том, что в то же самое время другой поток может вставлять новый

узел прямо после узла с числом 5 — узел, который будет безвозвратно потерян, поскольку _root ничего не знает о нем.

В литературе представлено несколько возможных решений; ни одно из них нельзя назвать тривиально простым. Реализация, представленная ниже, впервые была предложена Тимоти Харрисом в его работе с многообещающим названием «Прагматическая реализация неблокирующих односвязных списков» [30]. Эта реализация немного шероховата, поскольку ее логика основана на установке младшего неиспользуемого бита указателя _next. Идея состоит в том, чтобы сначала сделать на этом указателе пометку «логически удален» (обнулив его бит), а затем на втором шаге вырезать соответствующий узел целиком.

```
shared struct SharedList(T) {
   shared struct Node {
      private T _payload;
      private Node * _next;

      @property shared(Node)* next() {
         return clearlsb(_next);
      }

      bool removeAfter() {
         shared(Node)* thisNext, afterNext;
         // Шаг 1: сбросить младший бит поля _next узла,
         // предназначенного для удаления
         do {
            thisNext = next;
            if (!thisNext) return false;
            afterNext = thisNext.next;
         } while (!cas(&thisNext._next, afterNext, setlsb(afterNext)));
         // Шаг 2: вырезать узел, предназначенный для удаления
         if (!cas(&_next, thisNext, afterNext)) {
            afterNext = thisNext._next;
            while (!haslsb(afterNext)) {
               thisNext._next = thisNext._next.next;
            }
            _next = afterNext;
         }
      }

      void insertAfter(T value) {
         auto newNode = new Node(value);
         for (;;) {
            // Попытка найти место вставки
            auto n = _next;
            while (n && haslsb(n)) {
               n = n._next;
            }
            // Найдено возможное место вставки, попытка вставки
            auto afterN = n._next;
```

```
                newNode._next = afterN;
                if (cas(&n._next, afterN, newNode)) {
                   break;
                }
             }
          }
       }

       private Node * _root;
       void pushFront(T value) {
           .. // То же, что Stack.push
       }
       shared(T)* popFront() {
             // То же, что Stack.pop
       }
   }
```

Реализация непростая, но ее можно понять, если, разбирая код, держать в голове пару инвариантов. Во-первых, для логически удаленных узлов (то есть объектов типа `Node` с полем `_next`, младший бит которого сброшен) вполне нормально повисеть какое-то время среди обычных узлов. Во-вторых, узел никогда не вставляют после удаленного узла. Таким образом, состояние списка остается корректным, несмотря на то, что узлы могут появляться и исчезать в любой момент времени.

Реализации функций `clearlsb`, `setlsb` и `haslsb` грубы, насколько это возможно; например:

```
T* setlsb(T)(T* p) {
   return cast(T*) (cast(size_t) p | 1);
}
```

13.17. Статические конструкторы и потоки[1]

В одной из предыдущих глав была описана конструкция `static this()`, предназначенная для инициализации статических данных модулей и классов:

```
module counters;int counter = 0;
static this()
{
   counter++;
}
```

Как уже говорилось, у каждого потока есть локальная копия переменной `counter`. Каждый новый поток получает копию этой переменной, и при создании этого потока запускается статический конструктор.

---

[1] Описание этой части языка не было включено в оригинал книги, но поскольку эта возможность присутствует в текущих реализациях языка, мы добавили ее описание в перевод. – *Прим. науч. ред.*

```
import std.concurrency, std.stdio;

int counter = 0;

static this()
{
   counter++;
   writeln("Статический конструктор: counter = "  counter);

}

void main() {
   writeln("Основной поток");
   spawn(&fun);
}

void fun() {
   writeln("Дочерний поток");
}
```

Запустив этот код, получим вывод:

```
Статический конструктор: counter = 1
Основной поток
Статический конструктор: counter = 1
Дочерний поток
```

Объявить статический конструктор, исполняемый один раз при запуске программы и предназначенный для инициализации разделяемых данных, можно с помощью конструкции `shared static this()`**, а объявить разделяемый деструктор – с помощью конструкции** `shared static ~this()`**:**

```
import std.concurrency, std.stdio;

shared int counter = 0;

shared static this()
{
   counter++;
   writeln("Статический конструктор: counter = "  counter);

}

void main() {
   writeln("Основной поток");
   spawn(&fun);
}

void fun() {
   writeln("Дочерний поток");
}
```

В этом случае конструктор будет запущен только один раз:

```
Статический конструктор: counter = 1
Основной поток
Дочерний поток
```

Разделяемыми могут быть не только конструкторы и деструкторы модуля, но и статические конструкторы и деструкторы класса. Порядок выполнения разделяемых статических конструкторов и деструкторов определяется теми же правилами, что и порядок выполнения локальных статических конструкторов и деструкторов.

13.18. Итоги

Реализация функции `setlsb`, грязная и с потеками масла на стыках, была бы подходящим заключением для главы, которая началась со строгой красоты обмена сообщениями и постепенно спустилась в подземный мир разделения данных.

D предлагает широкий спектр средств для работы с потоками. Наиболее предпочтительный механизм для большинства приложений на современных машинах – определение протоколов на основе обмена сообщениями. При таком выборе может здорово пригодиться неизменяемое разделение. Отличный совет для тех, кто хочет проектировать надежные, масштабируемые приложения, использующие параллельные вычисления, – организовать взаимодействие между потоками по методу обмена сообщениями.

Если требуется определить синхронизацию на основе взаимоисключения, это можно осуществить с помощью синхронизированных классов. Но предупреждаю: по сравнению с другими языками, поддержка программирования на основе блокировок в D ограничена, и на это есть основания.

Если требуется простое разделение данных, можно воспользоваться разделяемыми (`shared`) значениями. D гарантирует, что операции с разделяемыми значениями выполняются в порядке, определенном в вашем коде, и не провоцируют парадоксы видимости и низкоуровневые гонки.

Наконец, если вам наскучили такие аттракционы, как банджи-джампинг, укрощение крокодилов и прогулки по раскаленным углям, вы будете счастливы узнать, что существует программирование без блокировок и что вы можете заниматься этим в D, используя разделяемые структуры и классы.

Литература

[1] Alagić, S., Royer, M. «Genericity in Java: Persistent and database systems implications». *The VLDB Journal*, том 17, выпуск 4 (2008), 847–878.

[2] Aldrich, J., Kostadinov, V., Chambers, C. Alias annotations for program understanding. Конференция «OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications», Нью-Йорк, 2002, ACM Press, с. 311–330.

[3] Alexandrescu, A. «On iteration». *InformIT* (ноябрь 2009). *http://erdani.com/publications/on-iteration.html.*

[4] Amsterdam, J. «Java's new considered harmful». *Dr. Dobb's Journal* (апрель 2002). *http://www.ddj.com/java/184405016.*

[5] Armstrong, J. «Programming Erlang: Software for a Concurrent World». The Pragmatic Programmers, 2007.

[6] Bacon, D. F., Strom, R. E., Tarafdar, A. «Guava: a dialect of Java without data races». Конференция «OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications», Нью-Йорк, 2000, ACM Press, с. 382–400.

[7] Benoit, F., Reimer, J., Carlborg, J. «The D Widget Toolkit (DWT)». *http://www.dsource.org/projects/dwt.*

[8] Bloch, J. «Effective Java Programming Language Guide». Sun Microsystems, Inc., 2001.[1]

[9] Bloch, J. «Effective Java. Second edition». Prentice Hall PTR, 2008.

[10] Böhm, C., Jacopini, G. «Flow diagrams, turing machines and languages with only two formation rules». *Communications of the ACM*, том 9, выпуск 5 (1966), 366–371.

[11] Boyapati, C., Lee, R., Rinard, M. «Ownership types for safe programming: Preventing data races and deadlocks». Конференция «OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications», Нью-Йорк, 2002, ACM Press, с. 211–230.

[12] Bright, W. «The D assembler». *http://digitalmars.com/d/1.0/iasm.html.*

---

[1] Джошуа Блох «Java. Эффективное программирование». – Лори, 2002.

[13] Brooks, Jr., F. P. «The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition». Addison-Wesley, 1995.[1]

[14] Cabana, B., Alagić, S., Faulkner, J. «Parametric polymorphism for Java: Is there any hope in sight?» *SIGPLAN Notices*, том 39, выпуск 12 (2004), 22–31.

[15] Cardelli, L. «Type systems». Глава 103 из книги «The Computer Science and Engineering Handbook», A. B. Tucker, Ed. CRC Press, 1997, с. 2208–2236.

[16] Chang, R. «Near speed-of-light on-chip electrical interconnects». Диссертация на соискание степени PhD, Стэнфордский университет, 2003.

[17] Cohen, T. «Java Q&A: How do I correctly implement the equals() method?» *Dr. Dobb's Journal* (май 2002). *http://www.ddj.com/java/184405053.*

[18] Digital Mars. dmd – FreeBSD D Compiler, 2009. *http://digitalmars.com/d/2.0/dmd-freebsd.html.*

[19] Digital Mars. dmd – Linux D Compiler, 2009. *http://digitalmars.com/d/2.0/dmd-linux.html*

[20] Digital Mars. dmd – OSX D Compiler, 2009. *http://digitalmars.com/d/2.0/dmd-osx.html*

[21] Digital Mars. dmd – Windows D Compiler, 2009 *http://digitalmars.com/d/2.0/dmd-windows.html.*

[22] Drepper, U. «What every programmer should know about memory». *Eklektix, Inc.* (октябрь 2007).

[23] Easton, W. B. «Process synchronization without long-term interlock». *ACMSIGOPS Operating systems review*, т. 6, вып. 1/2 (1972), с. 95–100.

[24] Findler, R. B., Felleisen, M. «Contracts for higher-order functions». *ACM SIGPLAN Notices*, т. 37, вып. 9 (2002), 48–59.

[25] Flanagan, C., Abadi, M. «Object types against races». «CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory», Лондон, 1999, Springer-Verlag, с. 288–303.

[26] Friedl, J. «Mastering Regular Expressions». O'Reilly Media, Inc., 2006.[2]

[1] Фредерик Брукс «Мифический человеко-месяц, или как создаются программные системы». – СПб: Символ-плюс, 2000.

[2] Дж. Фридл «Регулярные выражения», 3-е издание. – СПб: Символ-плюс, 2008.

[27] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. «Design Patterns: Elements of Reusable Object-Oriented Software». Addison-Wesley, 1995.[1]

[28] Gove, D. «Solaris application programming». PrenticeHall PTR, 2008.

[29] Gropp, W., Lusk, E., Skjellum, A. «Using MPI: Portable parallel programming with the Message-Passing Interface». MIT Press, Кембридж, 1999.

[30] Harris, T. L. «A pragmatic implementation of non-blocking linked-lists». *Lecture Notes in Computer Science 2180* (2001), с. 300–314.

[31] Herlihy, M. «Wait-free synchronization». *TOPLAS: ACM Transactions on Programming Languages and Systems*, т. 13, вып. 1 (1991), с. 124–149.

[32] Hoffman, D. M., Weiss, D. M., Eds. «Software fundamentals: collected papers by David L. Parnas». Addison-Wesley, 2001.

[33] ISO. The ANSI C standard (C99). Tech. Rep. WG14 N1124, ISO/IEC, 1999.

[34] Kernighan, B. W., and Ritchie, D. M. «The C Programming Language». Prentice Hall, 1978.[2]

[35] Knuth, D. E. «The Art of Computer Programming. Vol. 2: Seminumerical Algorithms» Addison-Wesley, 1997.[3]

[36] Korpela, J. K. «Unicode explained». O'Reilly Media, Inc., 2006.

[37] Lee, E. A. «The problem with threads». *Computer*, т. 39, вып. 5 (2006), с. 33–42.

[38] Liskov, B. «Keynote address – data abstraction and hierarchy». Конференция «OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications», Нью-Йорк, 1987, с. 17–34.

[39] Martin, R. C. «Agile Software Development, Principles, Patterns, and Practices». Prentice Hall, 2002.[4]

---

[1] Гамма Э., Хельм Р., Джонсон Р., Влиссидес Дж. «Приемы объектно-ориентированного программирования. Паттерны проектирования». – Питер, 2007.

[2] Керниган Б., Ритчи Д. «Язык программирования С», 2-е издание. Вильямс, 2008.

[3] Кнут Д. «Искусство программирования, т. 2: получисленные алгоритмы». – Вильямс, 2007.

[4] Мартин Р. «Быстрая разработка программ. Принципы, примеры, практика». – Вильямс, 2003.

[40] Meyer, B. «Object-Oriented Software Construction». Prentice Hall, 1988.[1]

[41] Meyers, S. «How non-member functions improve encapsulation». *C++ users journal*, т. 18, вып. 2 (2000), с. 44–52.

[42] Milewski, B. «Race-free multithreading: Ownership». Блог *http://bartoszmilewski.wordpress.com/2009/06/02/race-free-multithreading-ownership/*.

[43] Odersky, M., Wadler, P. «Pizza into Java: Translating theory into practice». 24 симпозиум «ACM SIGPLAN-SIGACT», по теме «Principles of programming languages», Париж, 1997, ACM, с. 146–159.

[44] Parnas, D. L. «On the criteria to be used in decomposing systems into modules». *Communications of the ACM*, т. 15, вып. 12 (1972), с. 1053–1058.

[45] Parnas, D. L. «A technique for software module specification with examples». *Communications of the ACM*, т. 15, вып. 5 (1972), с. 330–336.

[46] Pierce, B. C. «Types and programming languages». MIT Press,. Кембридж, 2002.

[47] Pike, R. «UTF-8 history». 2003. *http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt*.

[48] Press, W.H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P. «Numerical Recipes: The art of scientific computing». Cambridge University Press, Нью-Йорк, 2007.

[49] Radenski, A., Furlong, J., Zanev, V. «The Java 5 generics compromise orthogonality to keep compatibility». *J. Syst. Softw.*, т. 81, вып. 11 (2008), с. 2069–2078.

[50] Schmidt, D. C. «Strategized locking, thread-safe interface, and scoped locking». *C++ Report*, т. 11, вып. 9 (1999).

[51] Stepanov, A., Lee, M. «The Standard Template Library». Tech. rep., WG21 X3J16/94–0095, 1994.

[52] Sutter, H. «Virtuality». *C/C++ Users Journal* (сентябрь 2001).

[53] Sutter, H. «The free lunch is over: A fundamental turn toward concurrency in software». *Dr. Dobb's Journal*, т. 30, вып. 3 (2005), с. 202–210.

[54] Sutter, H. «Use threads correctly = isolation + asynchronous messages». Блог «Sutter's Mill» (март 2009). *http://herbsutter.com/2009/03/16/*.

[1] Мейер Б. «Объектно-ориентированное конструирование программных систем». – Русская редакция, 2005.

[55] Sutter, H. and Alexandrescu, A. «C++ Coding Standards: 101 Rules, Guidelines, and Best Practices». Addison-Wesley, 2004.[1]

[56] The Unicode Consortium. «The Unicode Standard, Version 5.0». Addison-Wesley, 2006.

[57] Valois, J. D. «Lock-free linked lists using compare-and-swap». 14 ежегодный симпозиум ACM по теме «Principles of distributed computing» (1995), ACM,. Нью-Йорк, с. 214–222.

[58] Von Ronne, J., Gampe, A., Niedzielski, D., Psarris, K. «Safe bounds check annotations». *Concurrency and Computation: Practice and Experience*, т. 21, вып. 1 (2009).

[59] Wadler, P. «Proofs are programs: 19th century logic and 21st century computing». *Dr. Dobb's Journal* (декабрь 2000).

[60] Wegner, P. «A technique for counting ones in a binary computer». *Communications of the ACM*, т. 3, вып. 5 (1960), с. 322.

[61] Xu, D. N., Peyton Jones, S., Claessen, K. «Static contract checking for Haskell». *SIGPLAN Notices*, т. 44, вып. 1 (2009), с. 41–52.

[1] Саттер Г., Александреску А. «Стандарты программирования на C++». – Вильямс, 2008.

Алфавитный указатель

Спецсимволы

Числа

A

K

L

M

N

O

P

R

S

K

Л

М

Н

О

Р

С

Марк САММЕРФИЛД

Qt. Профессиональное программирование

Разработка кроссплатформенных приложений на C++

560 стр., книга в продаже

Книга Марка Саммерфилда открывает путь к овладению разнообразными паттернами и приемами создания приложений с использованием библиотеки разработки кроссплатформенных приложений Qt.

Основной акцент сделан на создании моделей, графических представлений и гибридных приложений «рабочий стол + Интернет», на многопоточной обработке данных и приложениях, содержащих мультимедийные объекты и форматированный текст. Представлено подробное введение в подсистемы анимации и конечных автоматов, включенные в версию Qt 4.6.

В книге приведены примеры кода, протестированные на платформах Windows, Mac OS X и Linux с использованием Qt 4.6 (а многие работают также с версией Qt 4.5) и написанные с ориентацией на будущие версии Qt.

Хараламбос МАРМАНИС и Дмитрий БАБЕНКО

Алгоритмы интеллектуального Интернета

Передовые методики сбора, анализа и обработки данных

480 стр., книга в продаже

Если вас интересуют вопросы искусственного интеллекта, если у вас есть собственный блог, новостной портал, wiki-справочник или сайт с онлайн-игрой, если вы планируете создать веб-приложение, которое должно учитывать введенные каждым пользователем данные, его поведение в системе на протяжении некоторого периода времени, а также другую потенциально полезную информацию – эта книга будет вам безусловно необходима.

В издании рассматриваются проблемы, с которыми сталкиваются все веб-приложения: поиск, кластеризация, релевантность и т. д. Примеры в этой книге даны на языке Java и разработаны таким образом, чтобы с их помощью можно было проиллюстрировать универсальную методику – алгоритм, – которая находит применение в широком диапазоне сценариев.

Издание в первую очередь адресовано программистам и веб-разработчикам, однако множество примеров и новых идей будут полезны и руководителям разного уровня, желающим лучше разобраться в технологиях и их возможностях с точки зрения бизнеса.

Издательство "СИМВОЛ-ПЛЮС"

Основано в 1995 году

О нас

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.

Где купить

Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

Санкт-Петербург:

главный офис издательства –
В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 380-5007

Москва:

московский филиал издательства –
ул. 2-я Магистральная, д. 14В
(м. Полежаевская/Беговая),
тел. (495) 638-5305

Заказ книг

через Интернет http://www.symbol.ru

Бесплатный каталог книг высылается по запросу.

Приглашаем к сотрудничеству

www.symbol.ru

Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!

Андрей Александреску

автор неофициального термина «современный C++», под которым понимают множество полезных стилей и идей программирования на C++. Книга Александреску «Современное проектирование на C++: обобщенное программирование и прикладные шаблоны проектирования» полностью изменила методику программирования на C++ и оказала огромное влияние и на другие языки и системы. Благодаря многочисленным библиотекам и приложениям, разработанным Андреем, а также его исследовательской работе, он снискал уважение и практиков, и теоретиков.

С 2006 года Александреску стал правой рукой Уолтера Брайта – автора языка D и первого, кто взялся за его реализацию. Именно Андрей предложил многие важные средства D и создал большую часть стандартной библиотеки D. Все это позволило ему написать авторитетную книгу о новом языке D.

Цель языка программирования D – помочь программистам справиться с непростыми современными проблемами разработки программного обеспечения. Он создает все условия для организации взаимодействия модулей через точные интерфейсы, поддерживает целую федерацию тесно взаимосвязанных парадигм программирования (императивное, объектно-ориентированное, функциональное и метапрограммирование), обеспечивает изоляцию потоков, модульную безопасность типов, предоставляет рациональную модель памяти и многое другое.

Издание представляет собой введение в D, автору которого можно доверять. Книга в фирменном стиле Александреску – она написана неформальным языком, но без лишних слов и не в ущерб точности. Андрей рассказывает о выражениях и инструкциях, о функциях, контрактах, модулях и многом другом, что есть в языке D. В книге вы найдете:

- Полный перечень средств языка с объяснениями и наглядными примерами
- Описание поддержки разных парадигм программирования конкретными средствами языка
- Информацию о том, почему в язык включено то или иное средство, и советы по их использованию
- Обсуждение злободневных вопросов, таких как обработка ошибок, контрактное программирование и параллельные вычисления
- Таблицы, рисунки и «шпаргалки» – удобный справочный материал, незаменимый для практического решения задач с помощью D

Книга написана для практикующего программиста, причем она не просто знакомит с языком – это настоящий справочник полезных методик и идиом, которые облегчат жизнь не только программиста на D, но и программиста вообще.

Категория: программирование
Уровень подготовки читателей: средний

ISBN 978-5-93286-205-6