The D Programming Language

Andrei Alexandrescu

Addison-Wesley

HIGH TECH

Язык программирования D

Андрей Александреску

Символ®

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.

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: