Место для D

Андрей Александреску

D можно описать как высокоуровневый системный язык программирования.

Давайте посмотрим, почему язык D заслуживает более пристального внимания.

Конечно, я не пребываю в заблуждении, что смогу вас легко убедить. Мы, программисты, — странный народ. Достатчно взглянуть на то, как мы выбираем любимые языки и придерживаемся этих предпочтений в дальнейшем. Ожидаемая реакция программиста, заметившего на полке книжного магазина издание «Язык программирования XYZ» — «Даю себе ровно 30 секунд, чтобы найти в нём что-нибудь, что мне не понравится». Изучение языка программирования требует времени и усилий, а результаты появляются не сразу. Попытка избежать этого — проявление инстинкта выживания. Ставки высоки, вложения рискованны, так что лучше уметь принимать быстрое решение «против».

Изучение нового языка программирования может представлять интерес только в том случае, если язык удовлетворяет принципам, которые программист высоко ценит. Любое несовпадение создаёт у программиста впечатление небрежного и небезопасного или же навязчивого и нудного языка. Язык не может одновременно устраивать всех и каждого, он должен аккуратно выбирать основные координаты, позиционируя себя среди языков программирования.

Так что же особенного в этом D? Возможно, вы уже слышали о нём: языке с именем, похожим на не слишком остроумный каламбур; назойливо упоминаемый в новостных группах, посвящённых другим языкам; слишком часто расхваливаемый полным энтузиазма другом; или просто результат шального поискового запроса («Могу поспорить, есть неудачник, который разработал язык под названием D, посмотрим... О, смотри-ка!»).

Цель этой статьи - дать общее представление, так что некоторые особенности и концепции будут использованы без точного определения, если они интуитивно понятные.

Рассмотрим подробнее основные возможности языка. Заметьте, некоторые возможности и ограничения сформулированы неточно. Поэтому если что-то вас не радует, не спешите беспокоиться. Возможно, в следующем предложении вы найдете меняющее суть дела уточнение. Предположим, вы прочитали, что «D имеет сборщик мусора», и мурашки побежали у Вас по спине, тут же возникло желание плюнуть три раза через плечо и бежать прочь. А если Вы будете терпеливы, то узнаете что D имеет и конструкторы, и деструкторы, с помощью которых можно реализовать детерминированный срок жизни объектов.

Но прежде чем углубляться...

Прежде чем углубляться, хочу отметить кое-что, что вам следует знать. Прежде всего, если вы уже подумывали об изучении D, сейчас очень подходящее время заняться этим. В последнее месяцы D развивался стремительно, но в тишине, и многие потрясающие концепции становятся известны только сейчас, некоторые - из этой статьи. На момент ее написания, моя книга «Язык Программирования D» завершена на 40% и доступна для предзаказа на

Amazon. Несколько глав доступны здесь по основанной на подписках службе Rough Cuts Safari.

Существует две версии языка — D1 и D2. Эта статья рассматривает только D2. D1 стабилен (не будет подвергаться изменениям, за исключением исправления ошибок), D2 же пожертвовал обратной совместимостью, чтобы решить некоторые проблемы и добавить несколько важных элементов, связанных с параллельным и обобщённым программированием. При этом язык стал более сложным, что, на самом деле, является хорошим признаком. Ни один действительно используемый язык не становился меньше. Даже языки, разрабатываемые как «небольшие и элегантные» неминуемо растут по мере использования. Да, даже Lisp. Программисты, похоже, видят сны о небольших простых языках, но наяву хотят только бóльшую мощь для моделирования. Переходное состояние, в котором D пребывает в данный момент, ставит вас в незавидное положение человека, имеющего дело с «двигающейся мишенью». Я решил написать статью, которая описывает реализованные не полностью или вовсе находящиеся в разработке элементы, но зато не слишком быстро устареет.

Официальный компилятор D для всех популярных платформ (Windows, Linux и Mac) бесплатно доступен на digitalmars.com. В разработке другие реализации. Особо стóит отметить порт на .NET и компилятор, использующий инфраструктуру LLVM. Также существует две основных библиотеки — официальная, Phobos; и разрабатываемая сообществом Tango. Tango создавалась для D1 и всё ещё портируется на D2, а Phobos (версия которого для D1 была до безобразия мала и причудлива) претерпевает большие изменения и улучшения, использующие все преимущества D2. Неудивительно что споры о том, какая библиотека лучше, до сих пор в разгаре, но, похоже, конкуренция помогает обеим библиотекам быть настолько хорошими, насколько это возможно.

Наконец, что не менее важно, две библиотеки для создания графических интерфейсов дополняют возможности языка самым эффектным образом. Зрелая библиотека DWT — это непосредственный порт SWT. Более новая разработка — обвязка для использования необычайно популярной библиотеки от Qt Software для использования в D. Это не мелочь, учитывая что Qt — отличная (а, по мнению некоторых, лучшая) библиотека для разработки портируемых приложений с графическим интерфейсом. Эти две библиотеки полностью переносят D в измерение GUI. (*Существует также зрелая обвязка к библиотеке GTK+, не уступающей по качеству вышеназванным библиотекам — прим. переводчика*)

Основы D

Лучше всего D можно охарактеризовать как системный язык программирования высокого уровня. Он предоставляет возможности, которые обычно присущи языкам высокого уровня — такие как быстрый цикл итераций «исправил»-«скомпилировал»-«запустил», сборщик мусора, встроенные хеш-таблицы, или возможность пропустить множественные объявления типа — но так же и низкоуровневые особенности, такие как указатели, ручное управление памятью (*malloc/free* в C) или полуавтоматическое (использование, конструкторов, деструкторов, и уникального оператора scope), и в целом прямая работа с памятью, которую программисты C и C++ так любят. В действительности, в D можно связывать и вызывать C функции напрямую, без дополнительного уровня трансляции. Вся стандартная библиотека C доступна непосредственно из программ на D. Однако, вы редко почувствуете необходимость работать на таком низком уровне, потому что собственные средства D часто являются более мощными, более безопасными, и не менее эффективными. В целом, D показывает, что удобство и эффективность не обязательно противоречат друг другу. Не затрагивая тем более

высокого уровня, которые мы обсудим чуть позже, ни одно описание D не было бы полным, без упоминания внимательности этого языка к деталям: инициализируются все переменные, если их явно не инициализировать с помощью *void*; массивы и ассоциативные массивы интуитивно понятны и просты; итерирование не усложнено; NaN действительно используется; правила перегрузки легко понять; реализована встроенная поддержка документации и юнит-тестирования.

D — мультипарадигменный язык, который позволяет писать код в объектно-ориентированном, обобщенном, функциональном и процедурном стилях, а так же совмещать их в небольших участках кода. Следующих крохотный раздел дает некоторое представление о D.

Привет, больное место

Но давайте уже разберёмся с этим докучливым синтаксисом. Итак, без лишних церемоний:

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

Синтаксис чем-то похож на людскую одежду — рационально мы понимаем что он не делает особой разницы, но никак не можем прекратить обращать на него внимание (Я все еще помню девушку в красном из Матрицы). Для многих из нас D выглядит знакомым с его синтаксисом в стиле C, так же используемом в C++, Java и C#. (Я предполагаю, что вы знакомы с одним из них, поэтому мне не надо объяснять, что в D в порядке вещей целые числа, числа с плавающей точкой, массивы, циклы и рекурсия.)

Говоря о других языках, позвольте сделать удар по больному месту C и C++ версий программы «Hello, world». Классическая версия на C, известная из второго издания K&R, выглядит так:

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

и аналогичная классическая версия на C++ (отметьте рост энтузиазма):

```
#include <iostream>
int main()
{
     std::cout << "Hello, world!\n";
}
```

Множество сравнений вариантов этой программы на разных языках акцентирует внимание на длинне кода и количестве информации, необходимой для понимания примера. Давайте выберем другой путь и обсудим корректность, а именно: что произойдет, если, по какой-то причине, вывод в стандартный поток будет неудачным? Программа на C проигнорирует ошибку, потому что не проверяет результат, возвращаемый *printf*. По правде говоря, все несколько хуже; хотя на моей системе этот код компилируется и запускается, C возвращает неопределенное число в систему (На моем компьютере, завершается с кодом 13, что немного напугало меня. Тогда я разобрался почему: длина «*hello, world\n*» 13 символов;

printf возвращает количество напечатанных символов и помещает 13 в регистр EAX; при выходе к счастью не изменяется регистр; итак, в конечном счете 13 возвращается ОС как код возврата). Это значит что программа написана некорректно в рамках стандарта C89 или C99. После непродолжительного поиска в Internet можно найти более правильный вариант:

```
#include < stdio.h>
int main()
{
    printf("hello, world\n");
    return 0;
}
```

. . . что не сильно помогает в плане корректности, потому что непредсказуемый возврат заменяется на возврат кода успешного выполнения, независимо от того, успешно ли сообщение было выведено на экран на самом деле.

Программа на C++ гарантированно возвращает 0, если программист забывает вернуть значение, но тоже игнорирует ошибку, потому что. . . *std::cout.exceptions()* установлено в ноль, а значение *std::cout.bad()* никто после вывода не проверяет. Таким образом, обе программы возвращают код успешного выполнения, даже если они не смогли вывести сообщение по каким-либо причинам. Исправленные версии глобального приветствия на C и C++ теряют часть глянца:

```
#include <stdio.h>
int main()
{
   return printf("hello, world\n") < 0;
}
```

и

```
#include <iostream>
int main()
{
   std::cout << "Hello, world!\n";
   return std::cout.bad();
}
```

Дальнейшее расследование показывает, что классический «hello, world» для других языков, таких как Java (код пропущен из-за ограниченного размера статьи), J# (язык совершенно, совершенно не связанный с Java), или Perl, тоже всегда возвращает код успешного выполнения. Можно уже было бы подумать, что это заговор, но, к счастью, языки вроде Python и C# приходят на помощь, выбрасывая Exception.

А как с этим обстоят дела у D? Ничего изменять не потребуется. *writeln* выбрасывает exception при провале, а exception, выданный функцией main распечатывается в стандартный поток вывода ошибок (если возможно), и программа выходит с кодом ошибки. Короче говоря, всё, что надо, происходит автоматически. Я бы не стал бить по больному месту, если бы не две причины. Во-первых, забавно представлять себе уличные восстания миллионов преданых программистов, ноющих, что их «Hello, world» были подделкой. (Представьте слоганы: «Hello, world! Код возврата 13. Совпадение?» или «Доброе утро, стадо! Проснись!» и т.д.). Во-вторых, пример не отвлечённый, а показательный: D не только пытается заставить программиста писáть правильно, но и систематически пытается отождествить правильное решение и путь наименьшего сопротивления. Это возможно куда чаще, чем можно подумать. (И если вас тянет поправить мой код, «*void main()*» — это корректная конструкция

для D и делает именно то, что от неё можно ожидать. Языковым адвокатам, уничтожающим новичков, пишущих «*void main()*» вместо «*int main()*» в новостных группах, посвящённых C++, придётся найти другое времяпрепровождение, если они перейдут на D.)

Чёрт, я планировал обсудить синтаксис, а скатился в семантику. Возвращаясь к синтаксису, есть серьёзное отличие от C++, C# и Java: D использует форму записи вида *T!(X, Y, Z)* вместо *T<X, Y, Z>* (и *T!(X)* или просто *T!X* вместо *T<X>*) для описания параметризованных типов, и на это есть причины. Выбор угловых скобок, при их конфликте с арифметическими операторами „<», „»> и „>»>, был источником проблем при разборе программ на C++, приводивших к несметному числу особых правил и исключений, не говоря уж о малоизвестном синтаксисе *object.template fun<arg>()*. Если кто-то из ваших знакомых программистов C++ обладает самоуверенностью Супермена, спросите его, что делает этот код, и увидите Криптонит в действии. Java и C# также приняли угловые скобки, но мудро предпочли запретить арифметические выражения в качестве параметров, заранее лишаясь всех шансов на добавление их в язык в будущем. D расширяет традиционный унарный оператор „!» до бинарного и использует классические скобки (которые (я уверен) вы всегда сочетаете правильно)

Модель компиляции

Единица компиляции, защиты доступа и модульности в D — это файл. Единица упаковки (the unit of packaging) — каталог. Со сложностями на этом покончено. Вне всякого сомнения, исходный код будет лучше себя чувствовать в супер-пупер базе данных. Этот же подход использует «базу данных», отлаживаемую лучшими из нас в течении долгого времени, которая легко интегрируется с системами контроля версий, резервного копирования, защиты уровня ОС, журналирования, и что там у вас ещё есть. Кроме того, это образует низкий порог вхождения, так как всё, что нужно, чтобы начать разработку — текстовый редактор и компилятор. Буду откровенен, выбор специализированных средств разработки не слишком велик, но можно найти такие мелочи как d-mode для emacs, поддержка vim, плагин для Eclipse — Descent, отладчик для Linux — ZeroBugs, и полноценную IDE — Poseidon.

Генерация кода — классический цикл компиляции и компоновки, но он проходит намного быстрее, чем в большинстве подобных окружений по двум, нет, трём причинам.

- Во-первых, грамматика языка позволяет разделить и сильно оптимизировать лексический, синтаксический и семантический разбор.
- Во-вторых, вы можете приказать компилятору не создавать кучу объектных файлов, как поступает большинство компиляторов, а собирать всё в памяти, сделав всего одну линейную запись на диск.
- В третьих, Уолтер Брайт, создатель и первоначальный разработчик D, является неоспоримым экспертом в оптимизациях

Такое малое время отклика позволяет использовать D как интерпретатор (нотация shebang (#!) тоже поддерживается).

D имеет настоящую модульную систему, которая поддерживает раздельную компиляцию и автоматически создаёт и использует краткие сведения о модулях (умное словечко вместо «заголовочных файлов»), что избавляет от необходимости поддерживать лишние файлы самостоятельно, если только вам этого действительно не захочется, в этом случае вы можете. Я уже представляю как кто-то из вас в этот момент замолк на полуслове.

Модель памяти и многоядерность

Учитывая то, что D может вызывать функции C непосредственно, можно ожидать что D основывается на модели памяти C. Это было бы хорошими новостями, если бы не розовый слон в опасной близости от вазы династии Минь: многоядерные системы — архитектура с массовым параллелизмом, швыряющаяся в вас вычислительной мощностью так, будто бы она выходит из моды. Ах, если бы вы только были способны ею воспользоваться. Многоядерные системы уже здесь и способ, которым C разбирается с ними слишком зауряден и подвержен ошибкам. Другие процедурные и объектно-ориентированные языки в этом смысле немногим лучше, что привело к возврату интереса к функциональным языкам, которые полагаются на неизменяемость (immutability), элегантно избегая множества проблем, связанных с параллелизмом.

Появившись относительно недавно, D находится в завидном положении: он может учиться на ошибках предшественников. Видите ли, старомодные потоки работали так: после вызова примитива для запуска нового потока, он немедленно получал доступ ко всем данным в программе. При желании, с использованием специфических средств ОС, поток может получить доступ к так называемой локальной для потока области хранения данных. Короче говоря, по умолчанию память разделяется между всеми потоками. Это было источником проблем и в прошлом, а сейчас стало сущим адом. Причиной вчерашних проблем были неуклюжие обновления разделяемых переменных. Очень сложно отследить и должным образом синхронизировать обращения, не повредив данные. Но люди смирились с этим, потому что модель с общей памятью близка к аппаратной реализации и эффективна при правильной реализации. Вот мы и подобрались к «сущему аду» — сегодня память всё меньше и меньше разделяема между потоками. В настоящее время процессоры обращаются к памяти через глубокую иерархию, бóльшая часть которой локальна для каждого ядра. Таким образом, с разделяемой памятью не только трудно работать, оказывается, она становится всё более *медленной*, т.к. постепенно теряет связь с архитектурой оборудования.

Пока традиционные языки боролись со всеми этими проблемами, функциональные языки заняли принципиальную позицию, следующую из понятия математической чистоты. «Мы не заинтересованы в моделировании аппаратной части», сказали они, «Мы хотим смоделировать математику». А математика, в большинстве случаев, не имеет деструктивного присваивания и является инвариантом времени, что делает её идеальным кандидатом для параллельных вычислений. (Представьте момент, когда один из этих математиков, обращённых в программистов, услышал о параллельных вычислениях. Должно быть он хлопнул себя по лбу: «Погодите минутку!..») В кругах функциональных программистов быстро было замечено, что такая модель вычислений по своей природе предпочитает внеочередное и параллельное выполнение, но но до недавнего времени это было скорее скрытым потенциалом, нежели осознанной целью. Сегодня становится понятно, что функциональный, свободный от деструктивного присваивания стиль программирования будет отлично подходить хотя бы для некоторых частей любого серьёзного приложения, которое хочет использовать мощь параллельных вычислений.

Итак, где же во всём этом место D? Есть одна концепция, формирующая основу подхода D к параллелизму.

Память по умолчанию локальна для потока, становясь глобальной по явному запросу

В D память по умолчанию локальна для использующего её потока. Даже случайные глобальные переменные дублируются в каждом потоке. Когда требуется совместное владение

памятью, объекты могут быть объявлены со спецификатором shared, что делает их видимыми сразу нескольким потокам. Главное, что система типов знает о разделяемых данных и ограничивает набор выполнимых над ними операций, чтобы убедиться в правильном использовании механизмов синхронизации. Эта модель элегантно избегает множества опасных проблем, связанных с синхронизацией доступа в языках с общей по-умолчанию памятью. В этих языках система типов не представляет, какие данные должны быть разделяемы, а какие нет и слишком часто надеется на порядочность: она доверяет программисту подобающую аннотацию разделяемых данных. Потом возникают запутанные правила, призванные объяснить, что происходит в каждом отдельном случае, затрагивающем локальные данные, аннотированные общие данные, данные которые не аннотированы, но всё равно используются совместно, и любые комбинации вышеперечисленного. При этом всё это подаётся в настолько доступной форме, что те немногие, кто смогут в этом разобраться, будут в этом разбираться, а все остальные предпочтут не забивать себе голову.

Поддержка многоядерных систем — область очень активных исследований и разработок, и однозначно хорошая модель всё ещё не найдена. Начиная с прочного основания локальной по-умолчанию памяти, D шаг за шагом добавляет удобства, не грозящие ограничениями в будущем: чистые функции, неблокируемые примитивы, старое доброе программирование с захватами (lock-based programming), очереди сообщений (планируются) и многое другое. Более продвинутые возможности, такие как типы владения, обсуждаются.

Неизменяемость

Пока всё в порядке, но как насчёт математической чистоты неизменяемости и функционального кода? D осознаёт ключевую роль функционального программирования и неизменяемости в разработке надёжных параллельных программ (и не только параллельных, если уж на то пошло), поэтому в D для данных, которые никогда не изменяются, есть спецификатор *immutable*. В то же время, D понимает, что изменение данных часто является лучшим решением проблемы, не говоря уже о том, что к этому стилю мы все привыкли. Ответ D очень интересен: изменяемые и неизменяемые данные являются единым целым.

Почему неизменяемые данные это так здорово? Совместное использование неизменяемых данных потоками не требует синхронизации, а отсутствие синхронизации - самая быстрая синхронизация. Хитрость в том, чтобы быть уверенным: «только для чтения» и впрямь означает «только для чтения», а иначе нельзя дать никаких гарантий. Чтобы поддержать этот важный аспект параллельного программирования, D предоставляет беспрецедентную поддержку для смешивания функционального и императивного программирования. Данные со спецификатором *immutable* предоставляют сильные статические гарантии — правильно типизированная программа не может изменять *immutable* данные. Более того, неизменяемость транзитивна, и при работе на неизменяемой территории все ссылки тоже будут указывать на неизменяемые данные. (Почему? Если бы это было не так, то при совместном использовании потоками данных, которые вы считали неизменяемыми, вы непреднамеренно делаете общими и изменяемые данные, но в этом случае мы возвращаемся к сложным правилам которых хотели избежать). Целые подграфы связанных объектов могут быть легко «окрашены» неизменяемостью. Система типов знает где они и позволяет свободно разделять их между потоками, более агрессивно оптимизируя доступ к ним и в однопоточном коде.

D — не первый язык, предлагающий сделать глобальные данные локальными для потока. Что отличает D — это интеграция локальных для потока данных с неизменяемыми и изменяемыми данными в одной системе. Велико искушение рассказать об этом более детально,

но оставим это на будущее и продолжим обзор.

Безопасность — высший приоритет

Будучи системным языком программирования D разрешает очень эффективные и опасные конструкции: неуправляемые указатели, ручное управление памятью, приведение типов, которые могут разрушить самый аккуратный дизайн.

Но также имеет простой способ обозначить модуль «безопасным» и соответствующую модель компиляции, которая проверяет безопасность операций с памятью. Успешная компиляция кода в этом подмножестве языка — называемого «SafeD» — не гарантирует ни переносимость кода, ни то, что вы использовали только правильные методы программирования, ни отсутствие необходимости юнит-тестов. SafeD сфокусирован на устранении потенциальной порчи памяти. Безопасные модули (или безопасный режим компиляции) налагает дополнительные семантические проверки, которые запрещают использование опасных элементов языка, таких как арифметические операции с указателями или выведение переменных, находящихся на стеке за их область видимости.

В SafeD память не может быть испорчена. Безопасные модули должны составлять бóльшую часть большого приложения, тогда как «системные» модули должны быть очень редки и особенно тщательно проверяемы при пересмотре кода. Большинство хороших приложений могут быть написаны целиком на SafeD, но для чего-то вроде аллокатора памяти придётся засучить рукава и немного запачкать руки. Замечательно то, что не придётся использовать другой язык для реализации некоторых частей приложения. На время написания этой статьи SafeD не завершён, но находится в активной разработке.

Читайте по губам: лом больше не нужен

D — мультипарадигменный язык, что является претенциозным способом сказать, что вам больше не придётся всегда держать наготовое верный лом. D усвоил урок. Ничто не обязано быть объектом, функцией, списком, хеш-таблицей, или Зубной Феей. Всё зависит от того, что вы с ним делаете. Программирование на D может быть очень свободным, т.к. вам не придётся тратить время на поиск способов решения очередной проблемы с помощью вашего любимого лома. По правде говоря, свобода влечет за собой ответственность: теперь вам придётся тратить время, чтобы решить какой дизайн лучше подойдёт к решаемой проблеме.

Отказываясь от Единственного Правильного Способа, D следует традиции, начатой C++, с тем преимуществом, что D предлагает лучшую поддержку для каждой парадигмы, лучшую интеграцию между парадигмами и меньше проблем при следовании любой из них или всем сразу. Это преимущество хорошего ученика. Очевидно, что D многим обязан C++, так же как и менее эклектичным языкам, таким как Java, Haskell, Eiffel, Javascript, Python, и Lisp. (На самом деле большинство языков обязаны своим стилем Lisp'у, просто некоторые не хотят этого признавать.)

Хороший пример эклектичной натуры D — управление ресурсами. Некоторые языки ставят на то, что сборщик мусора — всё, что может понадобиться для управления ресурсами. Программисты на C++ уважают RAII, а некоторые считают что этого достаточно для полного управления ресурсами. Каждая группа не имеет опыта с инструментами, используемыми другой, что ведёт к смешным спорам, когда стороны даже не понимают аргументов друг друга. Истина в том, что ни один из подходов не является достаточным, поэтому D избегает фокусирования на чём-то одном.

Объектно-ориентированные возможности

В D есть как структуры, так и классы. У них много общего, но разное предназначение: к структурам обращение происходит по значению (value types), а классы предназначаются для динамического полиморфизма и обращение к ним происходит по ссылке (reference types). Таким образом, недоразумения и проблемы, связанные с обрезанием объектов, не существуют, как и комментарии вроде *// Нет! НЕ НАСЛЕДОВАТЬ!*. Когда вы разрабатываете тип, вы решаете заранее, будет он мономорфным value-типом, или полиморфным ссылочным типом. C++ разрешает определение двояких типов, но их очень редко используют из-за подверженности ошибкам и спорной ценности.

Объектно-ориентированность D предлагает возможности, похожие на Java и C#: одиночное наследование реализации, множественное наследование интерфейсов. Это делает код на Java и C# особенно легко портируемым на D. D предпочитает избежать поддержки множественного наследования на уровне языка, но так же не следует теории «Множественное наследование — это зло. Как вам может помочь оберег». Вместо этого D принимает сложность эффективной и полезной реализации множественного наследования. Чтобы предоставить преимущества множественного наследования за приемлемую цену D позволяет типу иметь несколько предков так:

```
class WidgetBase { ... }
class Gadget { ... }
class Widget : WidgetBase, Interface1, Interface2
{
    Gadget getGadget() { ... }
    alias getGadget this; // Widget наследует Gadget!
}
```

Alias действует так: когда ожидается *Gadget*, а всё, что у вас есть — это *Widget*, компилятор вызывает *getGadget*, чтобы получить его. Вызов полностью прозрачный, потому что если бы он не был прозрачным, это не было бы наследованием. Это было бы чем то разочаровывающе близким к наследованию. (Если вы увидели здесь намёк, скорее всего он есть.) Кроме того, *getGadget* имеет полную свободу действий в выполнении своей задачи. Например, он может возвращать компонент данного объекта, или новый объект. Вам всё равно придётся перехватывать вызовы методов, что может показаться огромным количеством рутинного кода, но здесь вступают возможности D по кодогенерации (читай дальше). Основная идея в том, что, используя *alias this*, D позволяет наследовать типы как вам угодно. Вы даже можете наследовать от *int*, если хотите.

D включает и другие проверенные техники, связанные с объектной ориентированностью, такие как явное использование ключевого слова override для избежания случайных перегрузок, сигналы и слоты, и техника, которую я не могу назвать из-за того, что она является зарегистрированной торговой маркой, поэтому назовём её контрактным программированием.

Функциональное программирование

Навскидку, как бы вы определили функцию получения числа Фибоначчи в функциональном стиле?

```
uint fib(uint n)
{
    return n < 2 ? n : fib(n − 1) + fib(n − 2);
}
```

Признаюсь, порой я предаюсь фантазиям. Одна из них — вернуться в прошлое и уничтожить эту реализацию функции Фибоначчи, чтобы ни один учитель информатики никогда никого ей не научил. (Следующими в списке являются сортировка пузырьком и реализация quicksort с пространственной сложностью $O(n*\log_2 n)$. Но ужастность *fib* легко превосходит их.) *fib* имеет экспотенциально возрастающее время выполнения, и как таковой не способствует ничему, кроме незнания сложности алгоритма и стоимости вычислений, мнения, что «симпатичному коду можно простить тормознутость» и вождения внедорожника в городе. Вы ведь понимаете насколько плоха экспоненциальность? *fib(10)* и *fib(20)* выполняется за пренебрежимо малое время на моей машине в то время как *fib(50)* требует девятнадцать с половиной минут. Скорее всего, вычисление *fib(1000)* будет длиться дольше, чем проживёт человечество, что меня утешает, потому что мы именно этого заслуживаем за то, что продолжаем учить плохому программированию.

Прекрасно. Так как же должна выглядеть «приличная» функциональная реализация Фибоначчи?

```
uint fib(uint n)
{
    uint iter(uint i, uint fib_1, uint fib_2)
    {
        return i == n
            ? fib_2
            : iter(i + 1, fib_1 + fib_2, fib_1);
    }
    return iter(0, 1, 0);
}
```

Эта версия требует пренебрежимо мало времени для вычисления *fib(50)*. Эта реализация сработает за $O(n)$,а оптимизация хвостовой рекурсии позаботится о пространственной сложности. Проблема в том, что новый *fib* потерял своё величие. В сущности эта реализация хранит две переменных состояния, замаскированных под аргументы функций, так что мы можем просто написать цикл, в котором это станет очевидным:

```
uint fib(uint n)
{
    uint fib_1 = 1, fib_2 = 0;
    foreach (i; 0 .. n)
    {
        auto t = fib_1;
        fib_1 += fib_2;
        fib_2 = t;
    }
    return fib_2;
}
```

Но (вопль ужаса) эта реализация больше не функциональная! Только посмотрите на все эти омерзительные присваивания в цикле! Одна ошибка и с вершин математической чистоты мы низвергаемся в бездну грязной массовой безыскусности.

Но если подумать минутку, итеративная версия *fib* не не так уж и нечиста. Если думать о ней как о чёрном ящике, *fib* всегда выдаёт одинаковый результат для заданных аргументов, а именно и это называется функциональной чистотой. Тот факт что в реализации содержится закрытое состояние делает его менее функциональным по букве, но не по духу. Осторожно

развивая эту мысль, мы приходим к очень интересному выводу: пока изменяемое состояние функции временно (другими словами, выделяется на стеке) и закрыто (не передаётся вместе со ссылками на функцию, которые могут его испортить) функция может считаться чистой (pure).

Именно так D определяет функциональнцю чистоту — функция может иметь изменяемое состояние, если оно временно и закрыто. Тогда вы можете написать pure в сигнатуре функции и компилятор без проблем её скомпилирует:

```
pure uint fib(uint n)
{
    // итеративная реализация ...
}
```

Такое облегчение чистоты очень полезно потому что вы получаете лучшее из двух миров: железные гарантии функциональной чистоты и комфортная итеративная реализация, когда это необходимо. Если это не круто, то я даже не знаю, чем вас ещё удивить?

Наконец, что не менее важно, функциональные языки имеют ещё один способ определения ряда Фибоначчи: так называемый бесконечный список. Вместо функции вы определяете ленивый бесконечный список, который будет отдавать всё больше чисел из ряда Фибоначчи по мере чтения из него. Стандартная библиотека D предоставляет достаточно интересный способ определения таких списков. Например, этот код выводит первые 50 чисел из ряда Фибоначчи (требуется импортировать *std.range*):

```
foreach (f; take(50, recurrence!("a[n−1]+a[n−2]")(0, 1)))
{
    writeln(f);
}
```

Это пример даже не в одну строчку, а в пол строчки! Вызов recurrence создаёт бесконечный список с формулой повторения $a[n] = a[n-1] + a[n-2]$ начиная с чисел 0 и 1. При всём при этом не происходит ни динамического выделения памяти, ни косвенных вызовов, ни расходования невосполнимых ресурсов. Код почти соответствует циклу в итеративной реализации. В следующей секции будет рассказано как такое возможно.

Обобщённое программирование

(Знакомо ли вам ощущение тревоги, которые вы испытываете, когда хотите рекомендовать приятелю фильм, книгу или музыку, которые вам действительно по душе и стараетесь не испортить всё излишней навязчивостью? Что-то подобное я чувствую, приближаясь к теме обобщённого программирования в D.) Обобщённое программирование имеет несколько определений, даже нейтральность статьи в Wikipedia обсуждается на момент написания этой статьи. Некоторые предпочитают думать об обобщённом программировании как о «программировании с параметризованными типами», тогда как другие имеют в виду «представление алгоритмов в наиболее обобщённом виде в котором они не теряют гарантий о сложности». Мы немного обсудим первое определение в этой части и второе — в следующей части.

D предлагает параметризованные структуры, классы и функции с простым синтаксисом. Например, вот функция *min*:

```
auto min(T)(T a, T b) { return b < a ? b : a; }
...
auto x = min(4, 5);
```

Где T — тип, a и b — обычные параметры функций. auto оставляет определение возвращаемого min типа на совести компилятора. Вот зародыш списка:

```
class List(T)
{
    T value;
    List next;
    ...
}
...
List!int numbers;
```

Веселье только начинается. Слишком многое относительно этой статьи нужно рассказать, чтобы оправдать затрагивание темы обобщённого программирования. Поэтому в следующих нескольких параграфах будут упоминаться только отличия от других языков, с которыми вы скорее всего уже знакомы.

Типы параметров. Не только типы могут быть параметрами шаблонов, но также и числа (целые и вещественные с плавающей точкой), строки, структуры определённые при компиляции и *синонимы* (aliases). Синоним — любая символическая сущность программы, которая, в свою очередь, может ссылаться на величину, тип, функцию, или даже шаблон. (Так D элегантно избегает бесконечной регрессии шаблонных шаблонных шаблонных ... параметров. Они просто передаются в виде синонима). Синонимы так же помогают определять лямбды. Списки параметров переменной длинны также разрешены.

Операции со строками. Передача строк в шаблоны была бы почти бесполезной если бы не было осмысленного способа оперировать над строками при компиляции. D предлагает полноценный набор операций (конкатенация, получение символа или подстроки, итерирование, сравнение ...).

Генерация кода: Assembler обобщённого программирования. Операции со строками при компиляции могут быть интересными, но они ограничены плоскостью данных. Чтобы перенестись в пространство необходима возможность преобразовывать строки в код (используя выражение *mixin*). Помните пример с *recurrence*? Он передаёт формулу продолжения ряда Фибоначчи в библиотеку в виде строки. Библиотека, в свою очередь, преобразует строку в код и отдаёт ей аргументы. В качестве ещё одного примера, вот так в D сортируются диапазоны:

```
// Определяем массив чисел
auto arr = [ 1, 3, 5, 2 ];
// Сортируем по возрастанию (по-умолчанию)
sort(arr);
// По убыванию, используя лямбду
sort!((x, y) { return x > y; })(arr);
// По убыванию, используя генерацию кода.
// Сравнение - обычная строка с параметрами a и b
sort!("a>b")(arr);
```

Генерация кода достаточно мощная. Она позволяет реализовывать компоненты без необходимости поддержки на уровне языка. Например, D не имеет битовых полей, но стандартный модуль *std.bitmanip* реализует их эффективно и полностью.

Интроспекция. В определённом смысле, интроспекция (возможность инспектировать элемент кода) — дополнение к генерации кода, потому что она позволяет смотреть на код,

вместо того чтобы генерировать его. Также она предлагает поддержку генерации кода. Например, интроспекция предоставляет информацию, необходимую для генерации функции разбора перечислимого значения. На данный момент, интроспекция поддерживается только частично. Запланирован улучшенный дизайн и реализация находится в приоритетном списке, следите за обновлениями.

is и static if. Любой, кто писал нетривиальный шаблон на C++ знает необходимость и трудности (а) определения, «скомпилируется ли» какой-нибудь код; и (б) статической проверки логических условий, и компиляции той или иной ветви кода. В D логическое выражение времени компиляции *is(typeof(expr))* возвращает *true* если *expr* — валидное выражение и *false* если нет (не останавливая компиляцию). Также, *static if* очень похож на *if*, кроме того что он работает на любом валидном логическом выражении при компиляции (иными словами, правильно сделанный *#if*). Я могу легко сказать что этих двух элементов достаточно чтобы уменьшить сложность обобщённого кода в два раза. И тот факт, что C++0x не имеет ни того ни другого преисполняет меня разочарованием.

Но подождите, это ещё не. . . Ну, вы поняли. Обобщённое программирование — огромное поле для игры и, несмотря на то что D покрывает его с удивительно компактным наоборм концепций, будет сложно обсуждать его не давая более подробной информации. D может предложить больше. Например, индивидуальные сообщения об ошибках, ограниченные шаблоны в стиле concepts C++0x (только немного проще. Пара порядков — ничто для друзей.), кортежи, а уникальная возможность которая называется «локальной конкретизацией» (необходима для гибких и эффективных лямбд) и, если вы позвоните в течении пяти минут, нож, который может разрезать замороженную банку пива.

Слово о стандартной библиотеке

Здесь я затрагиваю деликатный вопрос, потому что, как уже говорилось, существует две полноценные библиотеки, которые можно использовать с D — Phobos и Tango. Я работал только над первой, поэтому ограничусь её описанием. В моём восприятии, с тех пор как появилась STL, ландшафт библиотек контейнеров и алгоритмов изменился навсегда. На самом деле, он изменился так сильно, что любая библиотека разработанная после STL и не учитывающая её рискует выглядеть глупо. (Следовательно, своего рода совет, который я могу дать программистам на любом языке — научиться понимать STL.) Причина не в том, что STL — идеальная библиотека, это не так. Она безнадёжно привязана к преимуществам и недостаткам C++. Например, он эффективен, но имеет плохую поддержку высокоуровневого программирования. Симбиоз с C++ также делает её очень сложной для понимания программистами, незнакомыми с C++, потому что сложно увидеть суть через синтаксический хлам. Более того STL имеет свои дефекты, например, её концептуальный каркас не может должным образом вместить массу контейнеров и способов итерации по ним.

Главная заслуга STL в том, чтобы переосмыслить вопрос, что означает написать библиотеку фундаментальных контейнеров и алгоритмов и пересмотреть процесс написания таких библиотек. Вопрос, заданный STL: «Каков минимум того, что алгоритм может запросить у топологии данных, которыми он оперирует?» Удивительно, но большинство авторов библиотек и даже некоторые знатоки алгоритмов уделяли этой теме недостаточно внимания. STL обращает внимание на проблемы унифицированного интерфейса, в котором, например, обход массива и связанного списка можно унифицировать, а топологический аспект операции можно списать на детали реализации. STL показала недостаток такого подхода, потому что, например, глупо использовать унифицированный интерфейс только для того, чтобы

реализовать линейный поиск (если только вам не нравится ждать окончания выполнения квадратичных алгоритмов). Эти истины известны всем, кто имеет хотя бы малейшие познания в алгоритмах и их наиболее обобщённых реализациях в языке программирования. Несмотря на то, что я был знаком с основами алгоритмов, я вынужден признать, что никогда не задумывался о том, что такое чистый, Платонический линейный поиск, пока не увидел STL 15 лет назад.

Это я так ненавзячиво намекаю, что Phobos (смотрите *std.algorithm* и *std.range* в документации) во многом схож с STL. Я считаю, что набор алгоритмов, предлагаемых Phobos намного лучше, чем STL, по двум причинам. Во-первых, Phobos имеет очевидное преимущество в том, что он карабкается по спинам гигантов (не говоря о пальцах дварфов), во-вторых, он полностью использует возможности более мощного языка.

Диапазоны — хорошо, итераторы — плохо. Возможно, самое серьёзное отличие от STL в том, что в Phobos нет итераторов. Абстракция «итератор» заменена на абстракцию «диапазон», которая не менее эффективна, но предлагает значительно лучшие возможности для инкапсуляции, проверяемости и абстрагирующей мощности. (Если подумать, ни одна из примитивных операций над итераторами не проверяема. Это чудовищно.) Код, использующий диапазоны так же быстр, более безопасен и компактен, чем код, использующий итераторы. Больше никаких циклов for длиннее одной строки. Программирование с диапазонами настолько компактнее, что появляются новые идиомы, реализация которых была бы слишом громоздкой с помощью итераторов. Например, вам могла бы придти в голов функция *chain*, которая проходит по двум последовательностям, одной за другой. Очень удобно. Но *chain* принимала бы четыре итератора и возвращала два, что непрактично. *chain* с диапазонами, напротив, принимает два диапазона и возвращает один. Более того, вы можете использовать переменное количество аргументов, чтобы *chain* принимала любое количество диапазонов, и внезапно мы получаем очень полезную функцию. Функция *chain* уже реализована в модуле std.range. Например, вот пример итерации по трём массивам:

```
int[] a, b, c;
...
foreach (e; chain(a, b, c))
{
    // используем e . . .
}
```

Обратите внимание, что массивы не конкатенируются! *chain* оставляет их на месте и просто обходит по очереди. Это означает, что вы можете изменять элементы оригинальных массивов через chain. Например, попробуйте угадать что делает этот код: *sort(chain(a, b, c));* Вы совершенно правы, общее содержимое этих трёх массивов только что было отсортировано без изменения размеров оригинальных массивов так, что наименьшие элементы окаались в a и т.д. Это всего лишь маленький пример возможностей, предлагаемых диапазонами и комбинаторами диапазонов в сочетании с алгоритмами.

Ленивость до бесконечности и дальше. Алгоритмы STL (и многие другие) энергичные (eager): к тому моменту, как они возвращают значение, их работа уже завершена. Phobos, напротив, использует ленивые (lazy) вычисления там, где это целесообразно. Таким образом Phobos получает лучшие возможности для композиции и способность оперировать бесконечными диапазонами. Например, рассмотрим типичную высокоуровневую функцию *map* (популярна в кругах функциональных языков, не путать с одноимённой структурой данных STL) которая применяет данную функцию к каждому элементу в диапазоне. Если бы *map* не была ленивой, мы столкнулись бы с двумя проблемами:

- Во-первых, ей пришлось бы выделять память для хранения результата (список или массив).
- Во-вторых, ей пришлось бы пройти весь диапазон прежде чем вернуть управление вызывающей функции.

Первая проблема — проблема эффективности. Выделения памяти нужно избегать когда это возможно (например, вызывающая функция просто хочет посмотреть на результаты применения *map* по порядку). Вторая — проблема принципов. Энергичная реализация *map* просто не может обработать бесконечный диапазон, т.к. она окажется в бесконечном цикле.

Поэтому *map* в Phobos возвращает ленивый диапазон. Он вычисляет результат по мере потребления элементов из него. Функция reduce (противоположность *map* в своём роде), напротив, энергична. Некоторым функциям необходимы обе версии. Например, *retro(r)* возвращает диапазон, обратный данному, тогда как *reverse(r)* немедленно переворачивает сам *r*.

Заключение

Даже в обзоре есть ещё о чём поговорить, таком как юниттесты, UTF строки, выполнение функций при компиляции (что-то вроде интерпретатора D, работающего при компиляции программы), динамические замыкания, и ещё многом другом. Но если мне хоть немного повезло, вы уже заинтересованы. Если вы ищете системный язык программирования без агонизирующей боли, прикладной язык без скуки, принципиальный язык без характера, или — что важнее всего — взвешенную комбинацию всего вышеперечисленного, D может быть вам подходит.

Если вы хотите задать ещё какие-либо вопросы, пишите автору, а лучше настраивайте свои nntp приёмники на Usenet сервер news.digitalmars.com и пишите в группу digitalmars.d — сердце полного жизни сообщества.

Благодарности

Премного благодарен Scott Meyers, который указал на необходимость этой статьи и предложил название. Я получил отличные комментарии и предложения от Bill Baxter, Jason House, John «Eljay» Love-Jensen, Дениса Короскина, leonardo maffi (sic), Petru Marginean, Bartosz Milewski, Derek Parnell, Brad Roberts, Joel Salomon, Benjamin Shropshire, David Simcha, Florin Trofin, Cristian Vlasceanu, и Walter Bright.