Язык программирования D
(work in progress)

Бубненков Д.И.

2017

Оглавление

Глава 1

Введение

Если вам интересно системное программирование и вы планируете заниматься разработкой высокопроизводительных и масштабируемых приложений, то вполне возможно, что в настоящий момент одним из лучших, если не единственным выбором, будет язык программирования D.

Несмотря на то, что в последние годы появилось большое количество новых языков программирования, призванных упростить решение современных задач, таких как Go, Rust и Swift, D, пожалуй, является наиболее перспективным и универсальным. Так, к примеру, Go показывает себя крайне хорошо для написания микро-сервисов, но при этом практически полностью не пригоден для обработки данных и расчётов. Rust – является замечательным системным языком, однако имеет крайне высокий порог вхождения и зачастую неоправданно сложен, что делает его применения в больших проектах крайне неэффективным. Swift изобилует синтаксическим сахаром и является частью весьма закрытой инфраструктуры Apple.

В этом плане D выгодно отличается от других игроков на рынке, так как он является по-настоящему мультипарадигменным языком, что означает возможность разработки на нем как системных компонентов, так и прикладного программного обеспечения. Разумеется, вся эта мультипарадигменность имеет обратную сторону в виде определённого усложнения языка и повышения времени вхождения в него, однако обратной стороной этого является уверенность. На D вы сможете решить абсолютно любую задачу: начиная от разработки приложений для мобильных телефонов и встраиваемой микроэлектроники, заканчивая написанием движков СУБД, игр и различных высокопроизводительных веб-серверов.

Данная книга не является исчерпывающим руководством по языку, и главным образом нацелена на начинающих. Однако представленного материала будет достаточно для написания небольших и средних приложений размером в несколько тысяч строк кода. Предыдущий опыт программирования не обязателен, однако крайне желательно хотя бы поверхностное знакомство с любых из современных языков программирования.

Книга так же может быть полезна для тех, кто имеет опыт разработки на других языках. В таком случае она позволит сформировать общее представление о языке D, его синтаксических конструкциях, пакетном менеджере, а также устройстве стандартной библиотеки Phobos.

Среды разработки

D выгодно отличается от Java и C# тем, что разработку возможно вести практически в любом текстовом редакторе, имеющим подсветку синтаксиса. Это является следствием того, что в D отсутствует избыточный синтаксический сахар и огромное количество высокоуровневых обёрток, удаляющих программиста от понимания сути кода. Однако если вы предпочитаете использовать для разработки IDE, вы можете воспользоваться

одной из существующих. Однако в целях изучения языка автор рекомендует использовать текстовый редактор Sublime или NotePad++, т.к. они позволяют максимально полно сосредоточиться на написании кода.

Подготовка

Кроме текстового редактора нам потребуется так же компилятор, который можно загрузить на сайте dlang.org. Компилятор включает в себя пакетный менеджер `dub`, принципы работы которого мы рассмотрим в следующих главах.

Если вы работаете под Windows, то вы так же можете установить эмулятор командной строки `bash`. Его наличие не является обязательным, однако синтаксис команд в консоли будет приведет в соответсвие с его правилами. Различия будут незначительные. Так вместо команды `dir` будет использована аналогичная команда `ls`.

Перед началом работы мы договоримся о структуре каталогов. Это позволит значительно упростить работу. Каждый проект будет содержать папку `source`, в которую мы будем помещать наш код.

Часть 1. Первое приложение

Приложение Hello World

Наше первое приложение будет выглядеть следующим образом:

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

Как и все Си-подобные языки, D работает по тем же правилам. В начале вы указываете тип переменной или функции. Потом задаете ей имя. Если функция должна что-то принимать, то в круглых скобках указывается тип принимаемого значения и имя, которое будет использовано для него внутри самой функции. Ключевое слово `void` показывает, что функция ничего не возвращает.
Пример объявления переменной:

```
int x = 5;           + Присваиваемое значение
 ^   ^   ^           |
 |   |   |           |   +Имя переменной
 |   |   +-----------+   |
 |   |                   |   + Тип переменной
 |   +-------------------+   |
 |                           |
 +---------------------------+
```

Пример объявления функции:

```
      +--------------------------------+ возвращаемый тип
      |
      |        +-----------------------+ имя функции
      |        |
      |        |        +--------------+ тип принимаемых
      |        |        |                функцией данных
      |        |        |
      |        |        |      +-------+ имя, с которым мы будем
      |        |        |      |         использовать
      |        |        |      |         принятые данные внутри
      |        |        |      |         функции
      |        |        |      |
      v        v        v      v
   string myAnswer(string name)
   {
      return "Hello, " ~ name; //Символ ~ означает склейку строк
   }   ^       +-------------+
   ^   |             ^
   |   |             +------+ данные, которые вернет функция
   |   |                      после своей работы. В данном
   |   |                      случае текстовая строка
   |   |
   |   +--------------------+ ключевое слово, прерывающее
   |                          выполнение функции и возвращающее
   |                          данные
   |
   +-----------------------+ фигурные скобки ограничивают зону
                             видимости. Все, что объявлено
                             внутри них, не видно снаружи
```

Подробнее доступные типы мы рассмотрим в соответствующей главе.

Скопируем указанный код в текстовый файл, назовём его `app.d` и поместим его в папку `source`. После чего в терминале запустим команду для компиляции:

```
>dmd app.d
```

На выходе в том же самом каталоге, в котором находится файл с исходным кодом `app.d`, должен появиться скомпилированный файл приложения `app.exe`. Если вы запустите его, то на экране будет выведено `Hello World`. Обратите внимание, что если вы запустите ваше первое приложение вне сессии уже открытого терминала, то оно сразу же после своего запуска автоматически завершится, и вы скорее всего даже не успеете увидеть результат его работы. Это происходит потому, что код доходит до конца, и в конце происходит автоматическое завершение приложения. Чтобы этого не произошло, необходимо в конец вставить функцию ожидания ввода `readln();`, таким образом ваш код теперь должен выглядеть следующим образом:

```
import std.stdio;

void main() {
    writeln("Hello world!");
    readln();
}
```

В качестве альтернативы вы можете просто запускать скомпилированное приложение в уже открытой терминальной сессии, что позволит вам видеть в ней результаты работы вашего приложения.

Теперь давайте разберемся более подробно с тем, что происходит в нашем коде. `import std.stdio;` выполняет импорт модуля IO (Input/Ouput), в котором определен набор функций для ввода и вывода данных.

`void main() { }` является телом нашего приложения, или как его еще называют, точкой входа. Это значит, что при обращении к исполняемому файлу Операционная Система ищет в коде определённую сигнатуру, с которой и начинается выполнение приложения.

Концепция точки входа является базовой практически для всех языков программирования и среди компилируемых языков исключения крайне редки. Ключевое слово `void` означает тип возврата в операционную систему, и согласно стандарту D `main` всего должен возвращать тип `void`, хотя возможен возврат и других типов, к примеру `int`.

Именно внутри функции `main() { }` у нас происходит выполнение всех остальных функций и методов. Фигурные скобки `{ }` ограничивают зону видимости блока.

`writeln("Hello world!");` является функцией, которая в качестве аргументов принимает текстовую строку.

> **Tips:** Если вы до этого не занимались серьёзно программированием, то у вас может возникнуть некоторая путаница с тем, чем метод отличается от функции. Если вы ранее не встречались с концепцией ООП, то пока не дойдёте до соответствующей главы, постарайтесь представлять методы и функции схожими понятиями.

Язык D имеет встроенную поддержку Unicode, а значит строка `writeln("Привет Мир!");` является полностью корректной, однако если вы ее скомпилируете под Windows, то вы скорее всего увидите на экране кракозябры. Это происходит потому, что по умолчанию терминал Windows не работает с кодировкой Unicode и требуется переключить его вручную введя команду: `chcp 65001`. Однако не смотря на возможность этого, до сих пор правилом хорошего тона является использование латинского алфавита везде, где только можно, так как это позволяет избежать крайне большого количества всевозможных проблем.

Понятие о модулях

Единицей компиляции в D является модуль. Модуль представляет из себя файл, содержащий в себе логически объединённые блоки кода. В данном случае об App.d можно говорить как о главном модуле приложения, содержащем точку входа. Модули можно импортировать один в другой. Набор иерархически сгруппированных модулей называется `пакетом`. Пакет представляет из себя каталог на файловой системе, в котором находятся отдельные файлы-модули, объединенные в соответствии со своей функциональностью. В пакетах могут находиться вложенные пакеты, представляющие из себя папки с вложенными модулями.

По умолчанию D поставляется вместе с библиотекой Phobos, которая включает в себя весьма широкий набор модулей практически на все случаи жизни. Наряду с представленными в Phobos модулями программист может создавать свои собственные. Согласно правилам каждый модуль должен начинаться с заголовка, содержащего его имя (и имя пакета, если он находится внутри него).

Простейший пользовательский модуль выглядит следующим образом:

```
module foo;
import std.stdio;

void sayHello()
{
  writeln("Hello!");
}
```

Однако если вы попробуете скомпилировать указанный код командой:
`>dmd foo.d`
то вы получите ошибку, т.к. данный модуль не содержит точки входа, и операционная система просто не сможет понять с какого места следует начать его выполнять.

Правилом хорошего тона является давать модулю то же самое имя, что и сохраняемому файлу. То есть если вы дали модулю заголовок `module foo`, то будет правильно при сохранении дать ему имя `foo.d`.

Теперь после сохранение модуля `foo.d`, мы сможем получить к данным в нем из модуля, содержащего точку входа, путем вызова в начале `import foo;`. Обратите внимание, что все импорты работают исключительно внутри единичного модуля. Иными словами, если вы импортируете модуль `std.stdio` в модуле `app.d`, но не сделаете это в модуле `foo` , то вы получите ошибку компиляции. Поэтому необходимо импортировать библиотечные функции исключительно там, где вы планируете их использовать. Если вы добавите лишний импорт там, где он не нужен (в данном случае это в `app.d`), то ошибки не будет, однако делать это крайне не рекомендуется, так как это усложнит чтение кода. По этой же причине в D нет скрытого импорта, как к примеру в языке Python, где можно вызывать функцию `print()` не указывая из какой библиотеки ее брать.

Исходный код библиотеки Phobos расположен по адресу `dmd2/src/phobos/std`. Последняя часть пути `std` в данном случае является пакетом, все папки в данном каталоге – подпакетами, а файлы – пакетами.
Набор модулей библиотеки Phobos выглядит так:

- string.d
- file.d
- path.d
- conv.d
- …

При импорте модуля указывается каталог (имя пакета), в котором хранится модуль и через точку имя самого модуля. Таким образом если нам потребуется выполнить импорт модуля работы с файлами мы запишем: `import std.file;` без указания расширения `.d`. По умолчанию компилятор знает, где располагаются модули библиотеки Phobos, но пользовательские пакеты и модули следует располагать внутри каталога в главном модуле (`app.d`). Если нам потребуется выполнить импорт модуля `baz`, расположенного в подпакете `bar` пакета `baz`, мы просто укажем путь до него через точку: `import foo.bar.baz;`

По умолчанию все функции, методы и переменные являются публичными, т.е. при подключении модуля вы получаете полный доступ к его содержимому и можете вызывать определенные в нем функции, методы и переменные. Однако бывают случаи, когда модуль является достаточно большим, а вам нужно использовать из него лишь несколько функций. Импорт полного модуля может привести к конфликту имён. Эту проблему можно решить двумя способами.

1. Частичный импорт
2. Модификаторы видимости

Частичный импорт

Частичный импорт используется для функций и методов. Для частичного импорта после названия импортируемого модуля через двоеточие указывается, что именно мы хотим импортировать. К примеру, директива: `import std.stdio : writeln, readln;` импортируем из модуля `stdio` лишь две функции `writeln` и `readln`.

Модификаторы видимости

Модификаторы видимости служат похожим целям, однако позволяют кроме методов задавать зоны видимости как для переменных, а так же целых классов.

- `public` как было написано выше, по умолчанию все содержимое модуля является публичным, однако есть возможность это указать в явном виде.
- `package` содержимое видно только в пределах одного пакета или каталога.
- `protected` содержимое видно только в производных классах (расположенных в других модулях).

- `private` содержимое видно только внутри указанного модуля и к нему невозможно получить доступ из вне

Давайте теперь попробуем написать первое приложение, состоящее из двух модулей. Создадим папку с именем `source`, а в нее поместим два файла со следующим содержимым:

app.d:

```
import foo;
import std.stdio : writeln, readln;

void main()
{
  sayHello();
  writeln("Hello from main!");
  readln;
}
```

foo.d:

```
module foo;
import std.stdio : writeln;

void sayHello()
{
  writeln("Hello from module!");
}
```

Теперь для компиляции необходимо передать компилятору имена компилируемых файлов:

```
>dmd app.d foo.d
```

В результате у нас получится приложение с именем `app.exe`, которое после своего запуска будет выводить на консоль:

```
Hello from module!
Hello from main!
```

Обращаем ваше внимание, что в настоящий момент мы разрабатываем исключительно однопоточные приложения. Это значит, что все инструкции у нас будут выполняться последовательно и функция `writeln("Hello from main!");` будет вызвана исключительно после вызова `sayHello();` не зависимо от того, на сколько долго он будет выполняться.

Как вы могли заметить, созданная нами функция `sayHello();` вызывается с пустыми скобками, а библиотечная функция `readln;` без. Это происходит потому, что D позволяет опускать скобки в случае, если функция не принимает в себя никаких данных. Иными словами абсолютно корректно будет записать как `readln();`, так и

`sayHello;`. Однако для повышения читаемости кода имеет смысл использовать скобки.

D так же поддерживает концепцию UFCS (Universal Function Call Syntax). Это значит, что все функции могут быть вызваны по цепочке. То есть результат работы первой функции будет передан во вторую и т.д. Это позволяет значительно упростить чтение кода. Рассмотрим пример:

app.d:

```
import std.stdio;
import std.conv;

void main()
{
  getName.sayHello;
}

string getName()
{
  string name;
  write("Input username: ");
  name = readln;
  return name; //передаем данные в следующую функцию
}

void sayHello(string userName) //принимаем данные
{
  writeln("Hello, ", userName);
}
```

Полным эквивалентом будет вызов функции следующим образом:

```
...
void main()
{
  sayHello(getName);
}
...
```

Результат:

```
> app.exe
Input username: Mike
Hello, Mike
```

Обращаем ваше внимание, что функция `readln;` получает ввод с клавиатуры всех символов. Это значит, что при нажатии Enter во ввод будет передан специальный непечатаемый символ перевода строки. В этом легко убедиться, достаточно попробовать после запуска программы не вводить данные, а нажать клавишу Enter, а в самой программе добавить вывод размера переменной:

```
...
  name = readln;
  writeln("Length: ", name.length);
  return name;
...
```

Результат будет следующим:

```
> app.exe
Input username:
Length: 1
Hello,
```

Это может привести к проблемам, когда вы попытаетесь сравнить пользовательский ввод с заранее предопределенным значением и не смотря на кажущуюся идентичность код будет работать так, как если бы эти два символа были не равны. Чтобы избежать подобной ситуации для ввода с клавиатуры следует вызвать специальную функцию `chomp` из состава `std.string`, которая выкусывает из пользовательского ввода все непечатаемые символы. Для того, чтобы добавить ее, необходимо в начале файла `app.d` добавить `import std.string;`, а так же модифицировать функцию ввода следующим образом:

```
  write("Input username: ");
  name = readln.chomp; // выкусываем из ввода непечатаемые
символы
  writeln("Length: ", name.length);
```

> **Tips:** `writeln()` после вывода данных производит перевод курсора на строку ниже. Если это не требуется следует использовать функцию `write();`

Теперь после ввода пустого значения наше приложение как и ожидалось будет показывать, что значение переменной равно нулю.

```
> app.exe
Input username:
Length: 0
Hello,
```

Пакетный менеджер DUB

В состав компилятора D входит пакетный менеджер `dub`, который значительно упрощает управление зависимостями и позволяет гибко управлять правилами сборки приложения. В дальнейшем в целях упрощения мы будем использовать именно его.

Давайте посмотрим как он работает. Для начала командой `dub init` создадим заголовку проекта. В результате у нас будет создан каталог `source`, содержащий в себе файл `app.d`, а так же файл dub.sdl, в котором описываются настройки самого проекта. Теперь, если раньше при компиляции проекта, содержащего

в себе несколько модулей, нам было необходимо передать их все в качестве параметров для компилятора, то теперь достаточно поместить их в папку `source`, и дальше пакетный менеджер попытается скомпилировать все содержимое этой папки.

Для сборки проекта достаточно вызывать команду: `dub`, которая вначале выполнит компиляцию, а потом сразу произведет запуск скомпилированного файла. Если запуск не требуется, то можно выполнить команду `dub build`, которая произведет компиляцию без запуска.

> **Tips:** Все флаги компиляции D имеют свои аналоги в качестве параметров для сборки dub.

Теперь в случае необходимости подключения любой внешней библиотеки не нужно скачивать её отдельно и подключать. Достаточно в файле `dub.sdl` указать ее как внешнюю зависимость. Если библиотека зарегистрирована на `code.dlang.org`, то `dub` автоматически скачает и подключит ее, и программисту будет достаточно лишь выполнить ее импорт в нужном месте.

Теперь попробуем подключить библиотеку для работы с `ini` файлами: для этого в конец файла `dub.sdl` в качестве зависимости библиотеку `dini`. В результате файл настройки проекта должен выглядеть следующим образом:

dub.sdl:

```
dependency "dini" version="~>2.0.0"
```

После компиляции наше приложение будет помещено в папку самого проекта. В папке `source` должен располагаться исключительно код приложения и ничего больше. И в папке самого проекта приложение будет искать конфигурационный файл для чтения.

Подключим к файлу `app.d` необходимый импорт `import dini;`, а в папке проекта создадим конфигурационный файл `config.ini` со следующим содержимым:

```
[colors]
first=red
second=green
third=blue
[names]
name1=Mike
name2=David
```

Теперь нам останется только создать экземпляр класса [*] чтения конфигурационного файла и вызывать сам метод чтения конфига. В результате наш файл `app.d` должен приобрести следующий вид:

App.d:

```
import std.stdio;
import dini;

void main()
{
  auto ini = Ini.Parse("config.ini");
  writeln(ini["colors"].getKey("first"));
  writeln(ini["names"].getKey("name1"));
}
```

Обратите внимание, что в настоящий момент мы нигде не проводим обработку ошибок. Это значит, что если к примеру конфигурационный файл или имя секции, к которой мы обращаемся, будет отсутствовать, то наше приложение упадет с ошибкой.

После запуска `app.exe` мы увидим на экране следующие значения:

```
>app.exe
red
Mike
```

[*] Классы и методы будут объяснены в отдельной главе.

> **Tips:** 1. Минимальный `dub.sdl` составляет лишь одну строку и требует указания лишь имени: `name "app"` 2. Если вы хотите, чтобы скомпилированные файлы хранились не в корне проекта, а в отдельной директории `bin`, добавьте в `dub.sdl` следующую строку: `targetPath: "bin"`.

Базовые правила именования

Для упрощения чтения кода в D используется ряд соглашений по именованию.

1. Составные названия переменных, функций и имена экземпляров классов записываются в верблюжей нотации (camelCased). Пример: `getUserName`, `mySuperClass`. Одиночные с маленькой буквы/ Пример: `age`
2. Название классов, интерфейсов, структур, шаблонов, перечислений всегда должны начинаться с заглавной буквы. Пример: `MySuperClass`
3. Если первая буква акронима заглавная, то все остальные буквы так же должны быть заглавными. Есть строчная, то все остальные буквы так же должны быть строчными. Пример: `UTFException`, `asciiChar`.

Комментарии

В D используется те же правила комментирования, что и во всех Си-подобных языках. С небольшими дополнениями.
- одиночные строки комментируются символом `//`.
- множественные `/* some commented code */`.
- если вы планируете генерировать документацию к проекту на основании исходного кода, тогда для одиночной строки используется тройной слеш `///`,
- а для набора строк используется следующая последовательность символов: `/** some docs code */`.
- возможно так же использование вложенных комментариев, используя сочетание символов `/+ some commented code +/`
- если вы используете генератор документации, то комментарий должен располагаться строго до вызова метода или функции

Пример App.d:

```
import std.stdio;
import std.string;
/// Точка входа
void main()
{
  /// Эта строка не будет видна в сгенерированной документации
  sayHello("Mike");
  // Это простой комментарий, он не будет виден в документации
  foo();
}
/// Принимает имя пользователя userName с типом string
void sayHello(string userName) //
{
  writeln("Hello ", userName);
}

/**
Пример многострочного комментария.
Функция foo() ничего не делает.
*/
void foo()
{

}
```

Теперь запустим пакетный менеджер `dub` указав, что нам требуется сгенерировать документацию по проекту. Сделаем мы это следующей командой: `dub build --build=docs`.

Tips: Если вы хотите генерировать свежую версию документации после каждой сборки проекта, то имеет смысл добавить в `dub.sdl` строку: `dflags "-Dddocs"`. Для генерации документации без помощи `dub` просто вызовите компилятор с ключем `-D` и передайте ему список файлов, которые следует обработать. Пример: `dmd -D app.d foo.d`.

Система типов в D

D является строго типизированным языком. Это позволяет не только избежать огромного количества ошибок, которые порождает динамическая типизация, но и на этапе компиляции проводить целый ряд оптимизаций, позволяющих крайне значительно поднять скорость работы приложения. В D все типы переменных инициализируются значением по-умолчанию, это облегчает поиск возможных ошибок.

D является регистро-зависимым языком, так что с точки зрения компилятора переменные `myVar` и `myvar` имеют два разных имени. Имена переменных могут начинаться с подчеркивания и содержать в себе цифры (но не начинаться с них).

Ниже приведена таблица с имеющимися в языке D типами: (fixme).

Как уже было сказано выше статическая типизация позволяет провести ряд оптимизаций по скорости и потреблению памяти. Но в некоторых случаях, когда производительность кода крайне важна, мы можем дополнительно указать компилятору, что то или иное значение являться константой. Это позволяет провести дополнительные оптимизации.

```
int a = 1; // объявляем переменную a с типом int и присваиваем
ей в качестве значения число 1
immutable int b = 2; // объявляем неизменяемую переменную b и
присваиваем ей число 2
```

D так же поддерживает автоматическое определение типа переменной на основании присваиваемого ей значения. Для этих целей используется ключевое слово `auto`.

```
auto c = 3;
auto d = "my string";
```

В данном случае компилятор сам в состоянии определить, какой тип имеют данные. Однако злоупотреблять автоматической типизацией крайне не рекомендуется. Она значительно усложняет поддержку кода, и программисту на глаз становится практически невозможно понять. какой тип данных скрывается за ключевым словом `auto`.

Несколько слов нужно сказать о типе `size_t`. Данный тип был введен в Си-подобные языки с появлением 64-битных процессоров. Если во времена 32-битных процессоров тип `int` совпадал по размеру с размером указателя и был равен 32 битам, то с появлением 64-битных процессоров возникли определенные сложности с портированием программного обеспечения, т.к. размер указателя на 64-битных системах стал равен 64-битам. В итоге был введен новый беззнаковый тип `size_t`,

который на 64 битной. Иными словами `size_t` - это максимальный целочисленный тип для данной платформы. Если компилятор будет запущен на 128 битном процессоре `size_t` будет равен 128 битам.

Строковые типы

Следует отметить, что в таблице приведены только фундаментальные типы. Как вы могли заметить, в ней отсутствует, к примеру, тип `string`. Это происходит по той причине, что `string` является лишь обёрткой над массивом неизменных символов (`immutable(char)[]`). Неизменность символов означает, что вы не можете изменить отдельный символ в строке:

```
string str = "big";
str[1] = 'a';
```

Компилятор выдаст ошибку:

```
Error: cannot modify immutable expression str[1]
```

Строки в D представляют из себя структуру следующего вида:

```
struct {
  size_t length;
  <type>* ptr;
}
```

Где `length` - это длинна, а ptr - указатель на начало строки. Таким образом у каждой строки есть длинна и указатель на первый элемент. Давайте проверим это:

```
  string str = "abc";
  writeln(str);
  writeln(str.length); // размер строки
  writeln(str.ptr); // адрес первого символа
  writeln(*str.ptr); // получаем значение переменной по адресу
```

Результат:

```
> app.exe
abc
3
42E080
a
```

Первый элемент строки `abc` - это символ `a`, расположенный по адресу 42E080.

Как было сказано выше, строка- это набор неизменяемых `char []`. Это значит, что большинство операций, приводящих к изменению размера строки, будут происходить через дополнительное аллоцирование. То есть при изменении строки будет создаваться ее измененная копия. Главная причина подобного решения кроется в том, что если бы строки были изменяемыми, то это породило бы огромное количество ошибок в

многопоточных приложения, когда несколько потоков обращались бы к одной и той же строке, меняя ее произвольным образом.

Давайте теперь попробуем изменить строку и посмотреть, как изменится значение указателя на первый символ.

```
  string str = "abc";
  writeln(str.ptr);
  str = "def";
  writeln(str.ptr);
```

Вывод:

```
> app.exe
42E080
42E084
```

Как видим, адрес 42E080 увеличился на 4 бита и стал равен 42E084. Почему на? Дело в том, что для доступу к огромному количеству уже написанных библиотек на Си разработчики D были вынуждены добавить некоторые особенности языка Си. Хотя в D строки являются структурой, но в конце каждой из записей дописывается `0`, который в Си означает символ конца строки. Однако компилятор все это дело скрывает и для программиста на D строка - это строка.

Строки в D хранятся в виде массива известной длинны. Это позволяет не только избежать целого ряда ошибок, которые вызывают null-терминированные строки в C/C++, но и позволяет делать такую полезную вещь, как срезы. Срезы не приводят к дополнительной аллокации, т.к. представляют из себя лишь манипуляцию над указателями на начало и конец строки.

Тот факт, что строки представляют из себя структуры, имеющие размер и указатель на первый элемент, очень удобен тем, что появляется возможность делать над строками такой тип операции, как срезы - т.е. брать произвольное количество символов как с начала, так и с конца строки.

Предположим, нам требуется выбрать из строки символы с 2 по 5 (не забываем, что это мы считаем с единицы, а компилятор считает с нуля). Для этого мы напишем следующий код:

```
string str = "abcdefg";
string newstr = str[1..4];
writeln(newstr);
```

Результат работы кода будет следующим:

```
> app.exe
bcd
```

(checkme) В результате будет создана новая переменная, содержащая два указателя, ссылающиеся на исходную строку.

D из коробки поддерживает Unicode. Как вы знаете, в Unicode разные кодовые страницы разных алфавитов могут иметь разную длинну. Кроме того, некоторые символы могут иметь переменную длинну. Т.е. к примеру, в испанском слове сон `soñar` используется диакретическая литера `ñ`, которая кодируется как два символа: первым символом является английская буква n, вторым - надстрочный диакретический знак. Поэтому для хранения символов в кодировке Unicode следует использовать правильный тип.

В D предусмотрены три символьных типа:

- `char` размером 8 бит
- `wchar` размером 16 бит
- `dhar` размером 32 бит

Строки так же как отдельные символы имеют версии для различных Unicode кодировок. Так, строка `string` предназначена для хранения символов, равных 8 битам, `wstring` - 16, а `dstring` - 32. Если вы используете неправильный тип, к примеру, попытаетесь записать в переменную с типом `string` символы, часть из которых кодируется 16 битами, то ошибки не произойдет. Однако, если вы попытаетесь проверить размер данной переменной, то получите некорректный результат.

Пример:

```
string str = "soñar";
writeln(str.length);
```

Результат:

```
6
```

В то время, как в слове у нас всего пять букв. Однако, если мы используем тип `wstring`, то получим корректный результат.

Пример:

```
wstring str = "soñar";
writeln(str.length);
```

Результат:

```
5
```

Форматирование строк

Средства для форматирования строк в D крайне богаты, однако в повседневной жизни вам потребуется не так уж много. Для склейки строк у нас используется символ `~`.

Пример:

```
string str = "aaa" ~ "bbb" ~ "ccc";
writeln(str);
```

Результат:

```
aaabbbccc
```

Данные типа `char` всегда берутся в одинарные кавычки, тип `string` всегда в двойные.

Пример:

```
  char foo = 'a'; // одиночный символ
  string bar = "b"; // один или более символов
```

Во многих языках программирования большую проблему составляет вывод строк, содержащих спецсимволы, которые в некоторых случаях могут восприниматься как управляющие команды. К примеру, `\n` вызывает перевод текстового курсора на новую строку. Но как быть, если подобное сочетание встречается в обычной строке? К примеру, в пути: `Press yes\now for continue` или, к примеру, нам потребовалось бы вывести строку, содержащую в себе кавычки.

app.d:

```
void main()
{
  string str = "Press yes\now for continue";
  writeln(str);
}
```

Результат:

```
> app.exe
Press yes
ow for continue
```

Раньше проблему было принято решать экранированием всех проблемых символов, однако это значительно усложняло чтение кода, особенно в случае написания SQL запросов и путей в Windows, когда приходилось экранировать все кавычки и пути. Однако в D этой проблемы можно избежать путем обрамления строки, содержащей в себе подобные символы путем обрамления её символом апострофа (не путать с одиночной кавычкой):

```
string str = `Press yes\now for continue`;
```

Результат:

```
> app.exe
Press yes\now for continue
```

Еще одним способом представления строки является использование специальной литеры `q`. Эта литера, поставленная перед строкой, содержащей различные скобки, апострофы, кавычки, управляющие последовательности и т.д. позволяет представить их все в неизменном виде. Правила использования литеры `q` крайне простые. Вслед за ней

должен идти один из следующих открывающих и закрывающих последовательность символом: { }, [ ], ( ), < >. А между ними может располагаться любая строка.

Пример:

```
string str = q"<It's \0 \n>";
writeln(str);
```

Выведет:

```
> app.exe
It's \0 \n
```

Пример:

```
string str = q"[<Test>]";
writeln(str);
```

Выведет:

```
> app.exe
<Test>
```

Кроме этого при помощи все той же литеры q, только с несколько иным синтаксисом можно помещать в переменную целые куски кода, которые впоследствии можно использовать во время компиляции. На этот раз строка обязана открываться с символов q{ и закрываться символом }.

Главное правило как вы уже поняли такое, что для открывающей и закрывающей последовательности строки нужно использовать такой символ, который бы не встречался в самой строке и не воспринимался бы как ее завершение. Т.е. запись: `string str = q"[<Te]st>]";` является некорректной, т.к. в середине строки встречается закрывающий символ. Однако если перед ним будет использоваться открывающий символ, то строка будет полностью корректной: `string str = q"[<T[e]st>]";`

До этого в примерах использовали практически исключительно вывод при помощи функции writeln, которая после вывода строки совершает перевод текстового курсора на строку ниже. Если перевод не требуется, то следует использовать простую функцию write, однако в случае необходимости совершить перевод строки нам ничего не мешает внутри нее использовать все ту же служебную последовательность символов, указанную ваше: \n.

В некоторых случаях может потребовать расширенное форматирование строки так, чтобы можно было в специально отведённые места подставить значение переменных. Для этого используется функция writef и writefln соответственно. f от слова format.

Для каждого типа данных используется свой подстановочный символ. Для подстановки строкового типа %s, для целого числа %d, для числа с плавающей точкой %g. Полный

список подстановочных символов крайне велик и его всегда можно посмотреть в официальной документации.

app.d:

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

void main()
{
  string name = "Mike";
  int age = 20;
  float height = 184.5;

  writefln("Hi, %s you are %d, you height is %g", name, age,
height);
}
```

Вывод:

```
> app.exe
Hi, Mike you are 20, you height is 184.5
```

Чисто технически ничего не запрещает вам везде использовать строковый подстановочный символ `%s`. `writeln`, который в большинстве случаев сможет корректно привести тот или иной тип к строковому и вывести его. Однако в случаях, когда вам нужно, чтобы к примеру, тип с плавающей точкой был выведен до определённого количества знаков после точки, тогда рекомендуется указывать его тип в явном виде.

Обратите внимание, что только функции семейства `write` делают подстановку символов за вас. Если вам требуется передать строку с подстановками в другую функцию, то следует вызвать функцию `format`, форматирующую строку с учетом подстановочных переменных.

Пример:

```
...
string mylogin = "Mike";
string mypassword = "superpass";
string city = "London";

string sqlinsert = format(`INSERT INTO usersshapes (login,
password, cite) VALUES ('%s', '%s', '%s') `, mylogin,
mypassword, mycity);
stmt.executeUpdate(sqlinsert);
```

Формат представления данных в оперативной памяти

В современных операционных системах все приложения изолированы друг от друга с помощью так называемого механизма виртуальной памяти. Операционная система через специальные таблицы трансляции заставляет каждое приложение думать, что ему доступно все адресной пространство. Этот механизм был введен специально, т.к. иначе бы в многозадачной операционной системе приложения могли легко повредить память друг друга. Чисто технически существуют механизмы, которые позволяют приложениям обращаться к адресному пространству друг друга, но в данной книге мы их рассматривать не будем.

Говоря про память нельзя не затронуть такую важную часть, как указатели. Указатели представляют из себя специальные переменные, которые позволяют обращаться к участкам памяти. Объявляются они так же как переменные, но перед их именем ставится звездочка. Чтобы визуально отличать указатели от других переменных обычно к их названию добавляют буквы `ptr` от слова `pointer`. Тип, используемый при объявлении указателя, в точности должен соответствовать типу переменной, адрес которой мы присваиваем указателю.

```
int *x_ptr; // указатель на целое
string *str_ptr; // указатель на строку
```

Для получения адреса переменной используется символ асперсанда `&`. Не следует путать оператор взятия адреса со ссылкой на некоторое значение, которое так же визуально отображается символом `&`.

Как только мы взяли адрес переменной и поместили его в указатель, мы можем работать с указателем точно так же, как если бы он был значением переменной. Для того, чтобы превратить указатель в значение, необходимо выполнить операцию разыменования (перед указателем поставить звездочку).

```
  int x = 2;
  int *x_ptr = &x; // заносим в указатель адрес переменной 2
  writeln(x_ptr); //выводим адрес по которому лежит переменная 2
  writeln(*x_ptr); // преобразуем адрес в значение 2

  int z = *x_ptr+2; // тоже самое что x + 2
  writeln(z);
```

Ссылки и указатели часто используются при обработке данных, которые нет смысла передавать в функцию, т.к. операция копирования данных потребует не только дополнительные затраты по времени, но и по памяти.

Рассмотрим следующий код:

```
void main()
{
  int x = 2;
  calc(x);
  writeln(x);
```

```
}

void calc(int x)
{
  x = x+2;
}
```

Как несложно догадаться данный код выведет число `2`, т.к. функция `calc(int x)` принимает копию значения и работает уже с ним. Но как быть, если требуется изменить значение исходной переменной? Вот тут как раз оказываются полезны ссылки с указателями. Давайте исправим код так, чтобы в функцию мы передавали указатель на `x` и выполняли бы уже операции непосредственно с ним.

```
void main()
{
  int x = 2;
  calc(&x);
  writeln(x);
}

void calc(int *x) // тут происходит: *x = &x
{
  *x = *x+2; // к разымененнованному значению указателя
прибавляет 2 и кладем в него обратно
}
```

В результате строка `writeln(x);` выведет значение `4`.

Хотя указателями можно работать точно так же, как с простыми переменными, их следует использовать с крайней осторожность и только в том случае, если вы очень хорошо понимаете, что делаете. Неосторожное их использование может породить огромное количество ошибок обращения по неправильным адресам. Так, к примеру, вы можете удалить объект, на который был нацелен указатель, а указатель продолжит указывать на область памяти, где ранее располагались данные. В случае, когда программа записывает данные в память, используя такой указатель, данные могут незаметно разрушаться, что приводит к тонким ошибкам, которые очень трудно найти.

В большинстве случаев для того, чтобы работать с оригинальными данными, а не с их копией, достаточно использовать ключевое слово `ref`. Оно позволяет понять, что мы обрабатываем данные, расположенные по ссылке.

```
void main()
{
  int x = 2;
  calc(x);
  writeln(x); // x теперь равно 4
}
void calc(ref int x)
{
  x = x+2;
}
```

Результат:

```
4
```

Массивы

D поддерживает следующие типы массивов: статические, динамические, ассоциативные и строки, рассмотренные выше. Реализацию многомерных массивов мы затрагивать не будем.

Массив для себя проще всего представлять набором из элементов одинакового типа, у которых есть размер (`length`) и указатель (`ptr`) на первый элемент. Строго говоря, строка является частным случаем массива. Упрощено массив можно представить следующим образом:

```
struct {
  size_t length;
  T* ptr;
}
```

Где `T*` будет представлять из себя некий тип данных, который в случае со строкой был `char []`. Однако все остальные типы массивов по структуре будут несколько отличаться друг от друга.

Статические массивы

При объявлении статического массива программист заранее должен указать его размерность:

```
  int [5] x;
  x[4] = 3; // присваиваем пятому элементу (4-ому если считать от нуля) число 3
  writeln(x);
```

Результат:

```
> app.exe
[0, 0, 0, 0, 3]
```

Если вы попробуете обратиться в статическом массиве за его пределы, то компилятор выдаст вам ошибку.

```
  int [5] x;
  x[6] = 3;
Error: array index 6 is out of bounds x[0 .. 5]
```

Перечень свойств, доступных для статических массивов:

- `.init` значение по умолчанию
- `.sizeof` размер массива в битах (количество элементов умноженное на их размер)

- `.length` количество элементов
- `.ptr` указатель на первый элемент
- `.dup` создать динамическую копию текущего массива
- `.idup` создать динамическую копию текущего массива с типом immutable

Рассмотрим некоторые свойства.

init:

```
  int [5] x = [5,5,4,1,2];
  writeln(x.init);

  double [3] y = [1.4,25.01, 34.801];
  writeln(y.init);
```

Результат:

```
[0, 0, 0, 0, 0]
[nan, nan, nan]
```

length:

```
  int [5] x = [5,5,4,1,2];
  writeln(x.length);

  double [3] y = [1.4, 25.01, 34.801];
  writeln(y.length);
```

Результат:

```
> app.exe
5
3
```

```
  int [3] x = [1,2,3];
  int [] y = x.dup;
  y ~=4;
  writeln(y);
```

Результат:

```
> app.exe
[1, 2, 3, 4]
```

> **Tips:** Для выполнения различных манипуляций над массивами, таких как: сортировка, вставка эдементов и т.д. используйте библиотеку `std.algorithm`

Динамические массивы

В отличие от статических массивов размер динамических на этапе компиляции не известен и может меняться во время выполнения программы, и при обращении к свойству `length` мы всегда сможем получить точное количество элементов в массиве. В остальном динамические массивы имеют точно такой же набор свойств с тем лишь исключением, что свойство `.init` для них будет всегда равно `null`.

Следующий пример показывает расширение динамического массива.

```
int [3] x = [1,2,3];
int [] y = x.dup;
y ~=4;
writeln(y);
```

Операция ~= выполняет означает добавление элемента

Результат:

```
> app.exe
[1, 2, 3, 4]
```

Так же как и стороки массивы, имеющий один тип, можно склеивать:

```
int [] a = [1,2,3];
int [] b = [4,5,6];
int [] z;

z = a ~ b;
writeln(z);
```

Результат:

```
[1, 2, 3, 4, 5, 6]
```

Ассоциативные массивы

В разных языках этот термин называют по разному. В D под ассоциативными массивами понимают связку ключ-значение. Ключем и значением может быть любой тип.

После объявления формат записи значений в ассоциативный массив выгляди следующим образом:

```
aa[key] = value
```

Объявим ассоциативный массив с ключем в типа `int` и значением типа `string`:

```
string [int] aa;
aa[1] = "Mike";
aa[2] = "Jow";
```

```
  aa[3] = "David";

  writeln(aa);
```

Результат:

```
> app.exe
[3:"David", 2:"Jow", 1:"Mike"]
```

Ключем и значением может быть абсолютно любой тип данных. Давайте теперь сделаем, чтобы ключем и значением были строки:

```
  string [string] aa;
  aa["one"] = "Mike";
  aa["two"] = "Jow";
  aa["three"] = "David";

  writeln(aa);
```

Результат:

```
> app.exe
["three":"David", "two":"Jow", "one":"Mike"]
```

Значения в ассоциативных массивах можно перезаписывать:

```
aa["one"] = "Piter";
```

Изменит значение для ключа `one` на имя `Piter`. Если вы попытаетесь обратиться к несуществующему ключю, будет выкинуто исключение:

```
...
string str = aa["fourth"]; // пытаемся получить значение из
несуществюущего ключа
...
```

Удаление элементов производится с помощью функции `array_name.remove(key)`.

Пример:

```
  string [string] aa;
  aa["one"] = "Mike";
  aa["two"] = "Jow";
  aa["three"] = "David";

  writeln(aa);
  aa.remove("one"); // удаляем значение с ключем "one"
  writeln(aa);
```

Результат:

```
> app.exe
["three":"David", "two":"Jow", "one":"Mike"]
["three":"David", "two":"Jow"]
```

Ошибка: `core.exception.RangeError@app.d(48): Range violation`. Элемент, к которому вы пытаетесь, не существует.

Набор свойств у ассоциативных массивов значительно отличается от статических и динамических. Так, для них доступны следующие свойства:

- `.length` количество элементов
- `.keys` возвращает динамический массив, содержащий набор ключей
- `.values` возвращает динамический массив, содержащий набор значений
- `.byKey()` возвращает ленивый диапазон ключей без дополнительного выделения памяти
- `.byValue()` возвращает ленивый диапазон значений без дополнительного выделения памяти
- `.byKeyValue()` возвращает ленивый диапазон ключей и значений ассоциативного массива, к которым впоследствии можно обратиться как `.key` и `.value` без дополнительного выделения памяти
- `.rehash` переупорядочить элементы для повышения скорости. Полезно когда значения часто добавлялись и удалялись
- `.clear` очистить ассоциативный массив от содержимого. Для того, чтобы освободить занятую память, необходимо выполить `.rehash`
- `.get(Key key, lazy Value defVal)` проверяет, содержит ли массив указанный ключ, если нет, то возвращается значение по умолчанию

Давайте рассмотрим переборку ассоциативного массива по ключу и значению.

```
string [string] aa;
aa["one"] = "Mike";
aa["two"] = "Jow";
aa["three"] = "David";

foreach(element; aa.byKey())
{
  writeln("-> ", element);
}
```

Результат:

```
> app.exe
-> three
-> two
-> one
```

Если `foreach(element; aa.keys)` мы заменим на `foreach(element; aa.byValue())`, то на выходе получим не ключи, а значения.

Результат:

```
-> David
-> Jow
-> Mike
```

Для проверки существует ли ключ в ассоциативном массиве используется оператор `in`.

Давайте посмотрим как он работает:

```
  string [string] aa;
  aa["one"] = "Mike";
  aa["two"] = "Jow";
  aa["three"] = "David";

  if("somekey" in aa) // "somekey" в списке ключей у нас
отсутствует
  {
    writeln("somekey is exists in array");
  }
  else
  {
    writeln("somekey do not exists in array"); // поэтому
срабатывает этот блок
  }
```

Результат:

```
> app.exe
somekey do not exists in array
```

В некоторых ситуациях бывает проверить существует ли ключ в массиве и если нет, то вернуть какое-то другое предопределенное значение. Для этого используется свойство ассоциативного массива `.get()`. Давайте рассмотрим как оно работает.

Давайте проверим теперь как работает свойство `get()`.

```
  string [string] aa;
  aa["one"] = "Mike";
  aa["two"] = "Jow";
  aa["three"] = "David";

  string str = aa.get("two", "nothing"); // вернет "Jow"
  writeln(str);
  string str2 = aa.get("somekey", "nothing"); // вернет значение
по-умолчанию "nothing"
  writeln(str2);
```

Результат:

```
> app.exe
Jow
nothing
```

Ключ “two” существует и мы возвращем его значение в переменную `str`. Ключ был не найден “somekey”, поэтому переменная `str2` получила указанное значение по-умолчанию “nothing”.

Операторы языка

Логические операторы

Выше в тексте вы уже использовали один из самых логических операторов - символ равенства =, с помощью которого вы выполняли присвоения переменной определенного значения. Давайте рассмотрим какие логические операторы существуют еще.

== проверка на равенство. Если левая и правая часть совпадают, то будет возвращено значение `true`, в противном случае булет возвращено `false`.
`!=` проверка на неравенство. Если левая и правая части не совпадают, то будет возвращено значение `false`, в противном случае булет возвращено `true`.
`||` логическое или. Если хотя бы одно условие истина, то будет возвращено `true`, если нет, то `false`
`&&` логическое и. Если все условия истины, то будет возвращено `true`, если нет, то `false`
>, <, >=, <=. Операторы позволяют оценивать чистовые значения и возвращать `true` или `false` в зависимости от результата проверки.

Операторы можно группировать между собой и записывать сложные выражения. Пример:

```
  int x = 2;
  int y = 7;

if(x<3 && y>=5) // если `x` меньше трех И `y` больше или равен 5
  writeln("x is less than 3 and y is greater than 5");
else
  writeln("expression is false");
```

Результат:

```
 > app.exe
x is less than 3 and y is greater than 5
```

Для того, чтобы проверить булевые значения, достаточно просто записать их внутри оператора `if`.

Пример:

```
  bool myValue = true;
  if(myValue) // если myValue истина
    writeln("Block A");
  else // если myValue ложь
    writeln("Block B");
```

Результат:

```
> app.exe
Block A
```

Если требуется выполнить обратное действие – проверить значение на ложность, то в блоке `if` перед ее названием требуется поставить восклицательный знак.

Пример:

```
  bool myValue = true;

  if(!myValue) // если myValue истина
    writeln("Block A");
  else // если myValue ложь
    writeln("Block B");
```

Результат:

```
> app.exe
Block B
```

Операторы if и else

Операторы `if` и `else` позволяют проводить оценку условий и принимать решения о том, какая из ветвей кода должна выполняться.

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

void main()
{
  int x = 2;
  int y = 5;

  if(x<5)
    writeln("x is less than y");

  else
    writeln("x is greater than y");
}
```

Результат:

```
> app.exe
x is less than y
```

Форма записи блоков `if` и `else` без фигурных скобок, как в примере выше, является краткой. Если после условия `if` или `else` выполнить не единичной выражение, а блок кода, тогда необходимо обернуть его в фигурные скобки `{ ... }`.

Пример:

```
if(a<b)
{
  writeln("Block A"); // вызов первой функции
  foo(); // вызов еще одной функции
}

else
{
  writeln("Block B"); // вызов первой функции
  bar(); // вызов еще одной функции
}
```

Операторы циклов

Цикл for

Тело цикла `for` будет выполняться до тех пор, пока условие будет истинно (т. е. true). Цикл `for` всегда принимает три параметра: 1. стартовая переменная 2. условие работы 3. действие над стартовой переменной. Пример:

```
void main()
{
  int x;
  for(x=0; x<5; x++) // стартовое значение; условие работы;
действие над стартовой переменной
  {
    writeln(x);
  }
}
```

Результат:

```
> app.exe
1
2
3
4
```

Цикл foreach

Большинство современных программистов предпочитают использовать более высокоуровневый итератор циклов - `foreach`, который позволят не только писать меньше кода, но и сократить возможные ошибки, возникающие из-за неправильного указания значений итераторов. Цикл `foreach` еще называют умным `for`, т.к. он позволяет делать кучу вещей, для которых в цикле `for` пришлось бы писать много лишнего кода.

Синтаксис работы цикла крайне прост. В качестве первого параметра в теле цикла вы указываете произвольное имя перебираемой единицы, а вторым параметром указываете коллекцию, которую необходимо перебрать. `foreach` позволят одинаково успешно выполнять переборку огромного количества типов начиная с классических массивов и заканчивая структурами, кортежами, и даже сроки в файле. Простейший пример цикла эквивалентный циклу `for`, представленному выше:

```
void main()
{
  int [] myarr = [1,2,4,5];
  foreach(a;myarr)
    writeln(a);
}
```

Результат:

```
> app.exe
1
2
4
5
```

Точно так же цикл будет работать при переборки массивов строк. Пример:

```
void main()
{
  string [] names = ["Mike", "David", "Jow", "Elvis"];
  foreach(name;names)
    writeln(name);
}
```

Результат:

```
> app.exe
Mike
David
Jow
Elvis
```

Бывают ситуации, когда необходимо использование порядкового (индексного) номера перебираемого элемента. Для этого просто добавляем элемент счетчика первым

значением. Обычно для этого используется буква `i` от слова iterator, но может быть любая другая буква или набор символов.

Пример:

```
void main()
{
  string [] myarr = ["Mike", "David", "Jow", "Elvis"];
  foreach(i, a; myarr)
    writefln("i: %s, value: %s", i, a);
}
```

Результат:

```
> app.exe
i: 0, value: Mike
i: 1, value: David
i: 2, value: Jow
i: 3, value: Elvis
```

Не смотря на универсальность использовать `foreach` лучше для предназначенных для этого типов, которые являют собой понятие *коллекция*. В противном случае, если вы к примеру попробуете перебрать элементы класса, то возникнет резонный вопрос о том, в какой последовательности и по каким правилам нужно проводить переборку.

> **Tips:** На самом деле `foreach` является лишь оберткой над циклом `for`. Всю остальную работу компилятор делает за вас. При этом стоит отметить, что сам цикл `for` так же является оберткой над циклом `loop`, который и реализуется на уровне машинных команд.

Давайте попробуем прочитать текстовый файл и разбить его на строки по произвольному разделителю. В данном случае по запятой. Создадим текстовый файл `test.txt`, в который поместим строку: `Jow,David,Mike,Piter`. Не забудьте, что текстовый файл должен лежать там, где будет находиться исполняемый файл. При сборке проекта `dub` исполняемый файл всегда копируется уровнем выше каталога с исходным кодом `source`, а компилятор `dmd` оставляет скомпилированный файл рядом с файлом исходного кода. Если текстовый файл не будет найден приложение упадет с ошибкой, т.к. в примерах для сокращения кода мы подобные ситуации не обрабатываем.

app.d

```
import std.stdio;

void main()
{
  auto file = File("test.txt", "r"); // открываем файл на
чтение, передавая в качестве параметров имя файла и типом
доступа (r)ead.
  foreach(line; file.byLine(KeepTerminator.no, ',')) //вызываем
функцию разбивки по строкам и указываем ей тип разделителя
  {
```

```
    writeln(line);
  }

}
```

Результат:

```
> app.exe
Jow
David
Mike
Piter
```

Обратите внимание, что в данном случае мы указали в качестве параметров `KeepTerminator.no`, а значит сам символ разделителя в выводимых строках будет отсутствовать. Если нам требуется изменить поведение функции и заставить ее выводить символ разделителя, то просто заменим `KeepTerminator.no` на `KeepTerminator.yes`. Результат:

```
> app.exe
Jow,
David,
Mike,
Piter
```

[*] В данном примере рассматривается шаблонная функция. Если у вас пока нет опыта работы со структурами, классами и шаблонами, то код следует просто воспринять как должное. Более подробное объяснение этих вещей будет дано в соответствующих главах.

Как цикл `for`, так и `foreach` имеют поддержку операторов `break` и `continue`, первый выполняет выход из цикла, второй - пропускает шаг. Примеры их использования примедены в секции с описанием цикла `while`.

`foreach` поддерживает любой диапазон. Это значит, что мы можем использовать в его теле абсолютно любую функцию, которая возвращает данные с типом `range`. Рассмотрим пример использования библиотеки работы с диапазонами `std.range`. К примеру, нам требуется сделать переборку с данных с шагом 2. Т.е. обработать лишь каждый второй элемент.

Рассмотрим метод `stride` сигнатура вызова, которого следующая `auto stride(Range)(Range r, size_t n)`. `auto` - что он возвращает. Как было указано, использование слова `auto` хоть и позволяет писать более компактный код, но часто заставляет дополнительно вчитываться в описании метода или функции для того, чтобы понять, какой именно тип данных функция возвращает. В данном случае описание говорит нам: `Returns: At minimum, an input range.`.

Наверняка вы уже заметили, что большая часть функций в описании имеет два блока скобок: `foo(...)(...)`. Это шаблоны (шаблонные функции). Их поведение мы

рассмотрим в соответсвующей главе. Поэтому на первый блок скобок мы пока обращать внимание не будем, а сразу переключимся на второй `(Range r, size_t n).Range r` означает, что метод принимает некие данные с типом `range`(диапазон), `size_t n`, идущий вторым параметром означает, что ожидается целое число с типом (размерностью) `size_t`.

Мы помним, что в `foreach` вторым аргументом принимает диапазон, который как раз и вернет нам функция `stride`. Значит наш код будет выглядеть следующим образом:

```
import std.stdio;
import std.range; // подключаем модуль работы с диапазонами

void main()
{
  string [] myarr = ["Jow", "David", "Mike", "Piter", "Maria",
"Katerina"]; // помним, что строка это тоже range
  foreach(element; stride(myarr, 2)) // stride будет возвращать
каждый второй элемент
  {
    writefln("name: %s", element);
  }

}
```

Результат:

```
> app.exe
name: Jow
name: Mike
name: Maria
```

Обратите внимание, что только элементы в массиве типа `array` поддерживают индексный номер. Это значит, что если вы попробуете записать указанный цикл, добавив туда элемент счетчика `i`, то получите ошибку: `Error: cannot infer argument types, expected 1 argument, not 2`.

Для того, чтобы добавить к диапазону индексный номер, нужно применить к нему функцию `enumerate` со следующей сигнатурой: `auto enumerate(Enumerator = size_t, Range)(Range range, Enumerator start = 0)`. Второй блок скобок показывает, что функция принимает только один обязательный аргумент - диапазон `range`, знак = у второго параметра означает, что если его дополнительно не указать, то будет использовано значение по умолчанию - в данном случае 0.

В итоге, чтобы у нас нашего диапазона появился индексный номер, который бы работал внутри цикла `foreach`, мы должны записать следующий код:

```
import std.stdio;
import std.range; // подключаем модуль работы с диапазонами

void main()
{
  string [] myarr = ["Jow", "David", "Mike", "Piter", "Maria",
"Katerina"]; // помним, что строка это тоже range
  foreach(i, element; stride(myarr, 2).enumerate) // stride будет
возвращать каждый второй элемент
  {
    writefln("index: %s, name: %s", i, element);
  }

}
```

Результат:

```
> app.exe
index: 0, name: Jow
index: 1, name: Mike
index: 2, name: Maria
```

В результате оба варианта кода вернули нам каждый второй элемент в диапазоне.

Функционал модуля `std.range` крайне обширен и более подробно будет рассмотрен в отдельной главе.

Цикл while

Оператор `while` выполняет схожие функции с `for`, однако обычно используется для того, чтобы создать бесконечный цикл, который будет выполняться до тех пор, пока указанное в нем условие будет истино.

Пример:

```
import std.stdio;

void main()
{

  int i = 0;
  while(i<5) // условие выполняется пока `i` меньше 5
  {
    i++; // инкрементируем i на единицу на каждом шаге цикла
    writeln("i: ", i);
  }
  writeln("Outside while loop"); // цикл while уже завершился
}
```

Результат:

```
> app.exe
i: 0
i: 1
i: 2
i: 3
i: 4
Outside while loop
```

> **Tips:** Так же язык поддерживает оператор `do`, который работает по схожему принципу с оператором `while`. Однако в силу того, что он крайне реко используется в реальной жизни, мы не будем его рассматривать.

Оператор цикла continue

Бывают случаи, когда при работе с циклом требуется проверить какое-то условие, и если оно истино, то не выполнять следующий за ним блок кода, а сразу перейти к следующему шагу.

Пример:

```
import std.stdio;

void main()
{
  int i = 0;
  while(i<5)
  {
    i++; // инкрементируем i на единицу на каждом шаге цикла
    if(i==4) // если i равно 4
      continue; // пропустить шаг и не выполнять следующий код
    writeln("i: ", i);
  }
  writeln("Outside while loop"); // цикл while уже завершился
}
```

Результат:

```
> app.exe
i: 1
i: 2
i: 3
i: 5
Outside while loop
```

Как вы видите, в выводе отсутствует число 4, т.к. сразу после проверки цикл не стал выполняться дальше, а сразу перешел к следующему шагу и вывел число 5.

Оператор цикла break

Бывают случаи, когда при достижении какого-то условия нам не нужно выполнять блок кода дальше и требуется выйти из цикла. Для этого используется оператор `break`. В предыдущем примере давайте заменим оператор `continue` на `break` и посмотрим на результат.

```
import std.stdio;
void main()
{
  int i = 0;
  while(i<5)
  {
    i++; // инкрементируем i на единицу на каждом шаге цикла
    if(i==4) // если i равно 4
      break; // выходим из цикла
    writeln("i: ", i);
  }
  writeln("Outside while loop"); // цикл while уже завершился
}
```

Результат:

```
> app.exe
i: 1
i: 2
i: 3
Outside while loop
```

Оператор switch-case

Если оператор `if` используется для проверки какого-либо условия, то связка операторов `switch-case` обычно применяется там, где нужно проверить какую-либо переменную на совпадение с множеством возможных значением. Чисто технически ничего вам не мешает выполнять подобную проверку с использованием `if`, `else if`, `else`. В некоторых случаях код может получиться даже короче.

`switch-case` по своей сути лишь рудиментом от C/C++, который остался в D исключительно для упрощения процесса портирования кода. Польза в реальных проектах крайне низка. В современная CS (Computer Science) рассматривает паттерн-матчинг (сопоставление с образцом) как более передовой вариант блока `switch-case`. Концепция паттерн-матчинга пришла из функциональных языков программирования и позволяет не только писать меньше кода, но и выполнять сопоставление с различными типами данных, такими как: срезы, диапазоны, включать различные выражения в сопоставление и т.д. В настоящий момент паттерн-матчинг реализован лишь в нескольких языках, в число которых D к сожалению пока не входит, однако в дальнейшем такой функционал обязательно появится.

Правила использования лучше всего покажет следующий пример:

```
import std.stdio;
import std.range; // подключаем модуль работы с диапазонами

void main()
{
  string name = "Maria";
  switch(name)
  {
    case "Jow": // если имя совпало, вызываем функцию,
расположенную ниже
      sayHello(name); // вызываемая в случае совпадения функция
      break; // обязательно прерываем выполнение, т.к. искомое
значение найдено
    case "David":
      sayHello(name); // ...
      break;
    case "Maria":
      sayHello(name); // ...
      break;
    case "Piter":
      sayHello(name); // ...
      break;
    default: // обязательная секция, которая срабатывает, если
ни одно из сравнений не совпало
      writeln("Unknow name");
      break;
  }

}

void sayHello(string userName) // Принимает имя пользователя
userName с типом string
{
  writeln("Hello, ", userName);
}
```

Тот же самый код, записанный при помощи `if`, `else if`, `else`:

```
import std.stdio;

void main()
{
  string name = "Maria";

  if (name == "Jow")
    sayHello(name); // вызываемая в случае совпадения функция
  else if (name == "David")
    sayHello(name); // ...
  else if (name == "Maria")
    sayHello(name); // ...
```

```
  else if (name == "Piter")
    sayHello(name); // ...
  else
    writeln("Unknow name");
}

void sayHello(string userName) // Принимает имя пользователя
userName с типом string
{
  writeln("Hello, ", userName);
}
```

Оба варианта будут давать одинаковый результат:

```
> app.exe
Hello, Maria
```

Арифмитические операторы

+ сложение. Пример: `2+3`.
− вычитание. Пример: `3-2`.
++ инкремент (увеличение на единицу). Пример: `int i=2. i++`. Результат: `i` равно 3.
−− декремент (уменьшение на единицу). Пример: `int i=3. i--`. Результат: `i` равно 2.
* умножение. Пример: `2*3`. Результат: 6.
/ деление. Пример: `6/3`. Результат: 2.
^^ возведение в степень. Пример: `2^^3`. Результат: 8.

Если требуется получить результат в тот же самый оператор, из которого мы берем данные, то возможно использовать краткую запись. Предположим, что `i` у нас равно 5. Для того, чтобы прибавить к `i` число 2 и веруть результат в `i` можно записать как `i = i + 2;`, так и `i += 2;`. Обе записи будут абсолютно корректны.

Обратите внимание на размерность типов. Если вы попытаетесь записать в тип больше, чем он может в себя вместить, вы получите ошибку целочисленного переполнения. Как это выглядит? К примеру, тип `byte` позволяет хранить в себе как положительные, так и отрицательные числа от -128 до +127. Если вы к 127 прибавите единицу, то вместо ожидаемых 128 вы получите -128. То же самое с другими типами. В момент компиляции компилятор проверяет размерность данных, так что он просто не даст вам записать в переменную некорректное значение, однако он не проверяет значения, которые были высчитаны в процессе работы.

Рассмотрим пример:

```
import std.stdio;

void main()
{
  byte x = 127;
  while(x<=128) // условие не выполнится никогда и цикл будет
вечный
  {
    x++; // если бы x мог стать >129 то он бы не выполнялся
    writeln(x);
    readln; // для наглядности все делаем в пошаговором режиме
  }

}
```

Результат:

```
> app.exe
-128

-127

-126
...
-2

-1

0

1
```

Размеры целочисленны типов:

тип	размер	по-умолчанию
byte	-128 .. 127	0
ubyte	255	0
short	-32768 .. 32767	0
ushort	65,535	0
int	–2,147,483,648 .. 2,147,483,647	0
uint	4294967295	0
long	-9223372036854775808 .. 9223372036854775807	0L
ulong	18446744073709551615	0L

Количество поддерживаемых типов в D весьма велико, и полную их таблицу можно посмотреть в документации. Мы остановимся лишь на двух популярных.

Размеры дробных типов:

тип	размер	точность после запятой
float	1.17549e-38 to 3.40282e+38	6
double	2.22507e-308 to 1.79769e+308	15

Взаимодействие с операционной системой посредством Input/Output

Аргументы main

Существует большое количество способов передавать параметры в наше приложение, самый простой - при помощи аргументов командной строки, с которыми вызывается наше приложение. Самым простым примером будет передача аргуентов непосредственно в тело нашей точки входа `main()`.

Рассмотрим пример:

```
import std.stdio;

// приложение принимает массив строк, которые в теле будут под
именем args
void main(string [] args) // вместо args может быть использовано
любое другое слово
{
  writeln("You passed to app next arguments: ");
  foreach(arg; args[1..$])
  {
    writeln("Argument: ", arg);
  }
}
```

Теперь при запуске нашего приложения мы можем передать ему набор любых параметров через пробел, и все они будут выведены при его старте:

```
> app.exe -foo +bar baz
You passed to app next arguments:
Argument: -foo
Argument: +bar
Argument: baz
```

Обратите внимание, что при переборке значений в теле `foreach` мы используем конструкцию `args[1..$]`. Это слайс. Знак доллара означает последний элемент. В данном случае мы производим переборку начиная с первого элемента и заканчивая последним. Нулевой элемент мы в переборку не включаем. Это происходит по той причине, что по традиции нулевым аргументом является имя самого приложения, и если мы включим в выборку, то результат примет вид:

```
> app.exe -foo +bar baz
You passed to app next arguments:
Argument: app.exe
Argument: -foo
Argument: +bar
Argument: baz
```

Обычно таким образом передаются параметры, которые влияют на ход выполнения приложения. Чисто технически мы можем проверять эти параметры и на основании них менять ход выполнения приложения. Однако если ваше приложение будет принимать больше 1-2 аргументов, так лучше не делать, т.к. есть более специализированные варианты.

Парсинг аргументов командной строки при помощи getopt

`getopt` позволяет принимать аргументы командной строки, и делает это согласно определенным правилам. Обычно аргументы передаются в формате `--key=value`, возможно так же использовать краткую запись `-l=value`. Символом разделителя ключа и его значения так же может быть пробел: `--key value`, `-l value`.

Для работы `getopt` необходмо подключить модуль `std.getopt`. Сигнатура функции `getopt` следующая: `GetoptResult getopt(T...)(ref string[] args, T opts);`. Возвращает она тип `GetoptResult`, в качестве первого параметра принимает ссылку на входящие аргументы, принятые в `main`, в качестве второго значения аргументов. `T` означает некий абстрактный тип. Мы ведь можем принимать не только тип `string`, но и `int`, `bool`, `enum` и другие.

По умолчанию аргументы командной строки не чувствительны к регистру и ключи `--foo` и `--Foo` будут работать одинаково.

Для того, чтобы использовать однобуквенные аргументы, как к примеру, `-p` вместо `--password`, необходимо указать эти сокращенные версии через символ `|`. Пример:

Пример:

```
import std.stdio;
import std.getopt;

void main(string [] args)
{
  string login;
  string pass;

  getopt(args,
  "login|l", &login,
  "password|p", &pass);

  if (login == "admin" && pass == "superpass")
    writeln("You are logged as administrator");

  else if (login.length == 0 && pass.length == 0) // если логин И
пароль не указаны
    writeln("No argumens specified");
  else
    writeln("Credentials is invalid");
}
```

Теперь мы можем запускать наше приложение, передавая любые комбинации следующих аргументов:

```
> app.exe -l admin -p superpass
You are logged as administrator

> app.exe --l admin -p superpass
You are logged as administrator

> app.exe --login admin -p superpass
You are logged as administrator

> app.exe --login admin --password superpass
You are logged as administrator

> app.exe --login admin --password=superpass
You are logged as administrator

> app.exe --l=admin --password=superpass
You are logged as administrator
```

Несколько советов:
1. Если вам нужно установить булевое значение в `true`, то не обязательно писать `--writelog=true`, достаточно просто написать `--writelog`.
2. Если ключ (`string[] outputFiles;`) принимает массив значений (к примеру, набор имен файлов), тогда следует установить строковую переменную `arraySep`, указав ей значение разделителя и дальше передать через него набор входных значений. Пример:

```
string[] outputFiles;
arraySep = ",";  // По умолчанию раздилитель пробел, но он может создать лишнюю путаницу
getopt(args, "output", &outputFiles);
```

Теперь параметры можно задавать так `--output=myfile.txt,yourfile.txt` или так `--output myfile.txt,yourfile.txt`.

По умолчанию не включена обработка ошибок, так как если вы передадите программе неизвестный ключ, то она у вас упадет с исключением. Это сделано специально, т.к. дает программисту выбирать как должно вести себя приложение, если оно получило некорректные аргументы на входе. Однако существует возможность `getopt`, что он должен игнорировать все неизвестные ему аргументы. Для этого следует установить параметр `std.getopt.config.passThrough`.

Пример:

```
...
  getopt(args, std.getopt.config.passThrough,
  "login|l", &login,
  "password|p", &pass);
...
```

Теперь при передаче неизвестных параметров исключение вылетать не будет.

В некоторых случаях бывает полезно сделать некоторые ключи обязательными и информировать пользователя, что без их указания невозможно продолжить работу. Для этого используется параметр `std.getopt.config.required`.

Пример:

```
...
  getopt(args,
  "login|l", &login,
  std.getopt.config.required, "password|p", &pass); // пароль
обязателен
...
```

Теперь, если мы запустим наше приложение, указав только логин и забудем указать пароль, то получим исключение (которое, разумеется, следует обработать):

```
> app.exe -l Jow

std.getopt.GetOptException@C:\D\dmd2\windows\bin\..\..\src\phobos\std\g
etopt.d(727): Required option password|p was not supplied
```

Генерация help

Как вы заметили `getopt` возвращает структуру с типом `GetoptResult`, содержащую в себе информацию о всех возможных ключах, которые приложение рассчитывает получить в качестве возможных аргументов. Это бывает полезно для генерации справочной информации.

Для активации генерации справки явно писать обработку аргументов `--help|h` не требуется. `getopt` уже ожидает эти аргументы. Для того, чтобы вывести справку, достаточно лишь проверить булевый флаг `helpWanted` у переменной, содержащей в себе результаты `getopt`.

```
  if(result.helpWanted) // проверяем не был ли передан ключ `--
help|h`
    writeln("Help");
```

Пример:

```
import std.stdio;
import std.getopt;

void main(string [] args)
{
  string login;
  string pass;
  auto result = getopt(args,
  std.getopt.config.passThrough, // не падаем с исключением, если
переданы неизвестные аргументы
  "login|l", &login,
  "password|p", &pass); // получаем аргументы, но не
обрабатываем их

  if(result.helpWanted) // проверяем не был ли указан ключ --
help|h
  {
     defaultGetoptPrinter("Help: ", result.options); // Печатаем
доступные аргументы.
  }
}
```

Результат:

```
> app.exe -h
Help:
-l    --login
-p --password
-h     --help This help information.
```

Если мы хотим снабдить каждый ключ дополнительным коментарием, то после имени параметра просто укажем произвольный текст:

```
...
  auto result = getopt(args,
  std.getopt.config.passThrough, // не падаем с исключением,
если переданы неизвестные аргументы
  "login|l", "Login option discription", &login, // произвольное
описании параметра login
  "password|p", "Password should be at last 5 digits" ,&pass);
// произвольное описании параметра password
...
```

Результат:

```
> app.exe -h
Help:
-l    --login Login option discription
-p --password Password should be at last 5 digits
-h     --help This help information.
```

Для более удобного восприятия мы можем воспользоваться символом апострофа, описанным в разделе, посвященном форматированию строк, и обрамить наши комментарии к параметрам им так, чтобы в результате они выводились в кавычках.

Пример:

```
  auto result = getopt(args,
  std.getopt.config.passThrough, // не падаем с исключением,
если переданы неизвестные аргументы
  "login|l", `"Login option discription"`, &login,
  "password|p", `"Password should be at last 5 digits"` ,&pass);
// получаем аргументы, но не обрабатываем их
```

Результат:

```
> app.exe -h
Help:
-l    --login "Login option discription"
-p --password "Password should be at last 5 digits"
-h     --help This help information.
```

Обработка ошибок

Исключения

Во время исполнения приложения могут возникать ошибки двух типов. К первому типу относятся исключения (Exceptions). Ко второму ошибки (Errors). Исключениями называются ошибки такого рода, который программист может перехватить и каким-либо образом обработать. Исключения перехватывать можно и нужно. Ошибки фатальны и перехватить их **нельзя**. К примеру, если программа попытается прочитать отсутствующий файл, то не всегда будет правильно допускать падение приложение, особенно если этот файл не является необходимым для дальнейшей работы.

Исключения удобнее представлять как объект, которые содержит информацию о возникшей ошибке. В языке реализован специальный базовый класс исключений `std.exception`, от которого наследуются все остальные исключения. Правильно реализованная система исключений крайне важна, так как в больших проектах без нее становится практически невозможно найти ошибку. Именно поэтому большинство классов, выполняющие действия, которые могут привести к потенциальным ошибкам, имеют свой собственный набор исключений (унаследованный от базового класса). К примеру, класс, работающий с файловой системой, обычно имеет реализацию системы исключений, которая касается исключительно файловой системы. Класс, который работает с базой данных, реализует исключений для базы данных и т.д.

Нет никаких проблем выбрасывать исключения напрямую, унаследованные от `std.exception`, и многие библиотеки так делают, однако в некоторых случаях имеет смысл реализовывать и выбрасывать исключения уникального типа.

Рассмотрим пример самого простого исключения (пока рассматриваем исключительно реализацию базового класса исключений `std.exception`). Предположим, мы пишем приложение, обрабатывающее `.xml` файлы. И тут происходит неожиданная ситуация: пользователь пытается открыть в приложении файл `.png`. Библиотека (или модуль приложения, где происходит обработка) должна как-то проинформировать использующее ее приложение о том, что тип файла не соответствует ожидаемому, чтобы приложение само решило как быть в данной ситуации.

Кроме того, что функция должна выбросить исключение, вызывающий код его должен как-то поймать. Для поимки исключения используется блок `try-catch`. Блок `try` отвечает за нормальное выполнение, а блок `catch` за обработку исключения, если оно все же случилось в блоке `try`.

Пример:

```
import std.stdio;
import std.algorithm; // endsWith позволяет проверить конец
строки

void main()
{
  string filename = "myimage.png";
  try
  {
    write("My First Exception: ");
    processXMLFile(filename); // передаем имя файла в функцию,
ожидающую xml файл
  }

  catch(Exception e) // создаем объект исключения
  {
    writeln(e.msg); // обращаемся к полю, содержащему текст
исключения
  }
  finally
  {
    // не зависимо от того, поймати ли мы исключение или нет,
выполняем какую-либо операцию
  }

}

void processXMLFile(string name) // реализованная где-то в
библиотеке функция. Принимает имя файла
{
  if(!name.endsWith(".xml")) // если имя не заканчивается на
".xml"
    throw new Exception("Wrong file type"); // выкидываем
исключение
  // Продолжаем нормальный ход выпонения
}
```

Результат:

```
> app.exe
My First Exception: Wrong file type
```

Хорошей практикой при обработке исключений является использование блока `finally`, код в котором выполняется не зависимо от того, отработал ли блок `try` без ошибок, или вызываемый код выбросил исключение и управление было передано в обработчик `catch`. В блок `finally` принято помещать код, который необходимо выполнить и в том и в другом случае. К примеру. не зависимо от того, были ли отправлены данные в базу данных, нам надо закрыть к ней подключение.

Как было сказано выше, исключения делятся на отдельные типы в зависимости от того, что их породило. Была ли это ошибка файловой системы, ошибка обработки самого файла или ошибка в чем-то другом. Поэтому в сложных проектах принято писать отдельные обработчик `catch` для каждого типа ошибок. Обычно каждая библиотека снабжена документацией, в которой содержится перечень исключений с их типами, которые она может выбрасывать.

Выкидывание разных типов исключений вместо базового типа `Exception` полезно тем, что у программиста появляется возможность не только более точно понимать какого рода произошла ошибка в вызываемой функции, но и для каждого отдельного типа создавать свой обработчик. Это позволяет крайне гибко обрабатывать всевозможные ошибки. Технически нет проблем везде кидать базовый тип `Exception` вместо специализированного, однако это считается плохим тоном.

Обратите внимание. Исключения нужно кидать лишь когда возникает какая-то действительно нештатная ситуация и нужно проинформировать об этом код, который находится уровнем выше. К примеру, на уровне библиотеки, обрабатывающей `.ini` файлы, ошибка доступна к файлу или ошибка его структуры является нештатной ситуацией. Так что исключения нужно кидать только тогда, когда есть все логические основания считать, что код уровнем выше должен их ловить.

Как уже было сказано выше, значительна часть библиотек реализовывает свои классы исключений, унаследованные от базового `Exception`. К примеру, библиотека `std.stdio` выкидывает исключения с типом `StdioException`, `std.conv` с типом `ConvOverflowException`, `std.utf` с типом `UTFException` и т.д. Обычно документация содержит не только перечень функций библиотеки, но и типы исключений, которые она может выбрасывать.

В результате в коде для каждого типа исключений мы можем использовать свой собственный `catch` блок.

```
    try {
        // выполняем блок кода, который может выбрасывать
исключения

    } catch (first_exception_type)
    {
        // ловим первый тип исключений
        // делаем какую-то обработку
    }
    catch (second_exception_type)
    {
        // ловим второй тип исключений, который так же может
выбрасываться
        // делаем какую-то обработку
    }

    catch (Exception)  // ловим базовый тип исключений
    {
```

```
        // ловим второй тип исключений, который так же может
выбрасываться
        // делаем какую-то обработку
    }
    finally {
        // Выполняем действия не зависимо от того. были ли
брошены исключения или нет
    }
```

Каждый блок `catch` ловит исключительно тот тип исключений, который указан у него в параметрах. Так, если библиотека наряду с частными исключениями может выбрасывать какое-то общее с типом `Exception`, то непременно в самом конце нужно поймать и его.

В некоторых случаях обработки внутри блока `catch` бывает недостаточно и для того, чтобы проинформировать код, который находится уровнем выше, мы внутри блока `catch` можем выбрасывать исключение повторно.

Выглядит это следующим образом:

```
  try
  {
  // ...
  }

  catch(Exception e) // ловим исключение, случившееся в блоке
try
  {
    throw new Exception("Exception text"); // прокидываем его
выше
  }
```

Так как все исключения наследуются от базового класса `Exception` (точнее от его интерфейса), то во всех из них есть базовый набор свойств:
- `.file`: Имя файла, в котором произошло исключение
- `.line`: Номер строки откуда оно брошено
- `.msg`: Текст исключения
- `.info`: Состояние стека, когда исключение было брошено
- `.next`: Следующее исключение, расположенное за текущим

Атрибут nothrow

Бывают ситуации, когда некоторые функции не должны бросать исключения. Обычно это происходит в разных высококритичных к надежности программных компонентах, в которых любое непредвиденное поведение (а исключения являются именно таким случаем) просто недопустимо, т.к. невозможно на 100% быть уверенным, что никакие данные не были повреждены. То есть если ошибка все же возникла, то продолжать работу просто бесполезно. С этой целью используется атрибут `nothrow`. Которым помечается функция, после чего при любой попытке породить в ней исключение компилятор будет ругаться.

Пример:

```
import std.stdio;

void main()
{
  foo();
}

void foo() nothrow
{
  throw new Exception("Exception text");
}
```

Результат:

```
> dmd app.d
app.d(10): Error: object.Exception is thrown but not caught
app.d(8): Error: nothrow function 'app.foo' may throw
```

Блок scope

В D реализована так же поддержка так называемых `scope`. Они позволяют очень компактно записывать, что нужно сделать при выходе из блока кода. Синтаксис их крайне прост.

`scope(exit)` выполняется, если код вышел за пределы блока. Не зависимо, что было причиной: исключение или нормальное завершение `scope(success)` выполняется исключительно если блок был выполнен успешно `scope(failure)` выполняется если в блоке произошло исключение

Если за `scope()` выражением идет лишь одна строка, то фигурные скобки ставить не обязательно. Пример: `scope(exit) writeln("Will be written when exit");`

Если требуется выводить при помощи `scope()` большой блок кода, то она записывается в фигурных скобках.

```
scope(success) {
        writeln("Some text 1");
        writeln("Some text 2");
    }
```

Пример:

```
import std.stdio;

void main()
{
  writeln("Hello before");
  foo();
```

```
  writeln("Hello after");

}

void foo()
{
    scope(exit) writeln("1");
    scope(success) writeln("2");
    scope(exit) writeln("3");
    scope(failure) writeln("4"); // никогда не будет выведено
}
```

Результат:

```
> app.exe
Hello before
3
2
1
Hello after
```

А вот теперь давайте попробуем бросить в функции `foo()` исключение и посмотреть что будет:

```
import std.stdio;

void main()
{
  writeln("Hello before");
  foo();
  writeln("Hello after");
}

void foo()
{
    scope(exit) writeln("1");
    scope(success) writeln("2"); // никогда не будет выведено,
т.к. успех не равно ошибке
    scope(exit) writeln("3");
    scope(failure) writeln("4");
      throw new Exception("My Exception"); // кидаем исключение,
чтобы сработал блок failure
}
```

Результат:

```
> app.exe
Hello before
4
3
1

object.Exception@app.d(23): My Exception
```

Логирование

Большинство приложений как минимум в процессе своей отладки записывают возникающие ошибки в текстовый файл. Для этих целей Phobos реализован специальный модуль `std.experimental.logger`. `std.experimental.logger` позволяет не только записать ошибки в файл, но и посредством специальных флагов (`LogLevel`) дать возможность программисту указать какого рода ошибки он хочет перехватить. Это могут быть как безобидные информационные предупреждения, так и критические ошибки.

Для того, чтобы включить логирование, в приложении не обязательно создавать экземпляр класса `FileLogger`. В некоторых случаях можно использовать отдельные функции, которые имеют те же самые имена, что и методы класса `FileLogger`. Список функций:

```
log
trace
info
warning
critical
fatal
```

Имя файла, в который вызыванные функции будут записывать лог, можно установить, задав значение свойства `sharedLog` следующим образом:

```
sharedLog = new FileLogger("New_Default_Log_File.log");
```

Это свойство определяет куда именно выводить лог. Если его оставить (или снова сбросить) в значении по умолчанию (`null`), то лог будет выводится на консоль, а если установить, то в файл.

Несколько слов о том, что происходит в этом коде (подробнее мы поговорим про это в главе, посвященной ООП).

`sharedLog` является свойством (`@property`), которое можно установить точно так же как переменную. `new FileLogger("New_Default_Log_File.log");` создает экземпляр класса, в конструкторе которого устанавливается значение имени файла с логами, и это самое значение назначается на `sharedLog`.

Данное свойство должно быть задано уровнем выше вызываемых функций так, чтобы они имели доступ к его значению.

Если же вы создаете экземпляр класса, то в его параметрах при создании вы указываете желаемое имя лог-файла. Пример:

```
auto fLogger = new FileLogger("NameOfTheLogFile.log");
```

Рассмотрим более подробно, как это выглядит на практике. В первом случае мы установим значение `sharedLog` и будем писать наши логи в файл.

```
import std.stdio;
import std.experimental.logger;

void main()
{
  sharedLog = new FileLogger("New_Default_Log_File.log");

  log("Log message: ");
  info("Info message: ");
  warning("Warining: ");
  error("Error: ");
  critical("Critical error: ");
  //fatal("Fatal error: ");
}
```

На консоль у нас не будет выведено ничего, зато в файле “ появятся следующие записи:

```
2017-01-30T15:38:20.073:app.d:main:8 Log message:
2017-01-30T15:38:20.073:app.d:main:9 Info message:
2017-01-30T15:38:20.073:app.d:main:10 Warining:
2017-01-30T15:38:20.073:app.d:main:11 Error:
2017-01-30T15:38:20.073:app.d:main:12 Critical error:
```

Обратите внимание, что у всех без исключения указанных функций есть версии с поддержкой форматирования. Точно так же, как у `writeln` есть аналог `writefln`. То есть если вам требуется в сообщение лога вставить какое-то значение из кода просто напишите: `logf("Log message: %s", x);`, где `x` будет строка со значением, определенная где-то выше.

Если того требуется, вы можете создать полноценный экземпляр класса `FileLogger` и пользоваться его методами. Выглядеть это будет следующим образом:

```
import std.stdio;
import std.experimental.logger;

void main()
{
  auto fLogger = new FileLogger("TestLogFile.log");

  fLogger.log("Log message: ");
  fLogger.info("Info message: ");
  fLogger.warning("Warining: ");
  fLogger.error("Error: ");
  fLogger.critical("Critical error: ");
  //fLogger.fatal("Fatal error: ");
}
```

В итоге будет создан лог-файл `TestLogFile.log`, абсолютно идентичный предыдущему.

Разница между двумя способами - вызовом отдельных функций и созданием экземпляра класса заключается в двух моментах:
1. Экземпляр класса создает новый неймспейс, и у вас появляется возможность использовать другую функцию с именем, к примеру, `log` (при условии, что вы не импортировали функцию `log` при подключении `std.experimental.logger`)
2. При использвании экземпляра класса у вас есть возможность унаследоваться от `Logger` и переопределить его поведение.

Tuples

Иногда бывает нужно упаковать несколько разнотипных значений в один объект. Для этой цели используются кортежи (Tuples). Кортеж представляет из себя нечто среднее между структурой и массивом. С одной стороны элементы в нем могут иметь различный тип как в структуре, с другой - у них есть индексный нормер как в массиве. Некоторые типы являются кортежами, к примеру `DicEntry`, содержащий записи о директории является ни чем иным, как кортежом.

```
alias DicEntry = Tuple!(string, string); // имена можно опустить
```

Для работы с кортежами необходимо подключить библиотеку `std.typecons`. После чего создать кортеж можно будет двумя способами.

1. Вызывать функцию `tuple` и передать в нее данные как аргументы
2. Вызывать конструктор типа `Tuple` и передать в него набор шаблонных аргументов и сами данные:

```
import std.stdio;
import std.typecons;

void main()
{
  auto myTuple = tuple(42, "Hello"); // вызываем функцию,
создающую кортеж
  writeln(myTuple); // выводим структуру кортежа
  writeln(myTuple[0]); // выводим первый элемент кортежа
  writeln(myTuple[1]); // выводим второй элемент кортежа

  auto myTuple2 = Tuple!(int, string)(56, "Privet"); // используем
конструктор типа
  writeln(myTuple2);

}
```

Результат:

```
> app.exe
Tuple!(int, string)(42, "Hello")
42
```

```
Hello
Tuple!(int, string)(56, "Privet")
```

При создании кортежа есть возможность дать его элементам имя для того, чтобы впоследствии можно было бы обращаться к ним по имени, а не по индексу. Имена можно задавать только путем указания их в конструкторе после каждого из типов.

Делается это следующим образом:

```
import std.stdio;
import std.typecons;
void main()
{
  auto myTuple2 = Tuple!(int, "number", string, "mytext")(56,
"Privet");
  writeln(myTuple2.number);
  writeln(myTuple2.mytext);
}
```

Результат:

```
> app.exe
56
Privet
```

Часто кортежи используются для того, чтобы вернуть из функции сразу несколько значений.

Пример:

```
import std.stdio;
import std.typecons;
void main()
{
  auto result = foo();
  writeln(result);
}

auto foo()
{
  int a = 2;
  int b = 3;
  string c = "ccc";
  // ...
  auto myTuple = tuple(a,b,c);
  return myTuple;
}
```

Результат:

```
> app.exe
Tuple!(int, int, string)(2, 3, "ccc")
```

Глава 2

Объектное Ориентированное Программирование

Классы

В предыдущей главе мы уже немного касались темы ООП. Сейчас мы рассмотрим ее более подробно.

Главной задачей ООП является упростить разработку больших и сверхбольших проектов. ООП само по себе не является серебрянной пулей и его применение не всегда бывает полезно. Поэтому D не навязывает эту концепцию как единственно правильную. Если сильно захотеть, то можно писать программы вообще без ООП или с его частичным применением.

Строго говоря, код у программиста - лишь побочный продукт. Чем его меньше, тем лучше. Главная задача состоит в решение проблем, а уж как эти проблемы будут решаться - зависит от каждого конкретного случая. D не поощрает написание классов где нужно и не нужно, как это делает Java и C#. Напротив, D позволяет не писать их попусту, тем самым значительно сокращая код и упрощая его поддержку.

В предыдущей главе мы уже косвенно затрагивали тему классов и экземпляров классов.

По своей сути класс - это шаблон реального объекта, а экземпляр - сам реальный объект, с которым идет работа. Каждый класс включает в себя набор полей и методов для работы с ними (функции, расположенные внутри класса, принято называть методами). Не стоит забывать, что многие функции в D не являются членами какого-либо класса. Такие функции называются просто функциями.

Важно понимать, что класс является лишь шаблоном. Каждый раз, когда мы будем выполнять какое-либо действие, мы будем работать с так называемым экземпляром класса.

Гораздо проще эту концепцию показать на примере.

Класс выглядит следующим образом:

```
          +---------------+ ключевое слово,
          |                 обозначающее класс (шаблон класса)
          v
       class MyClass <-----+ имя класса
       {
+--->   string mydata; <--+ поле класса, которое мы
|                          инициализируем ниже
|
|       this(string mydata) <--+ данные, которые мы
|                          принимаем из конструктора класса
|       {
+-----+   this.mydata = mydata; <-+присваиваем полю
                                  класса данные из конструктора
          }

          void sayHello(string name) <------+ функция,
                                    расположенная внутри класса,
                                    называется методом
          {
            writeln("Hello, ", name);
          }

          // выполняем действия над mydata
       }
```

Итак, мы описали наш первый класс. В нем находится одно поле `mydata` и один метод `sayHello()`, принимающий текстовую строку. Конструкция `this( ... )` называется конструктором класса. Она обычно принимает при создании класса какие-то данные и на основе этих данных производит инициализацию полей. Внутри самого конструктора `.this` используется для устронения неоднозначности в случае, если входные данные и поле класса имеют одинаковые имена.

Для того, чтобы классом можно было пользоваться, его нужно реализовать. Т.е. создать его экземпляр, который будет иметь точно такую же структур,у как и оригинальный класс, но в отличие от шаблона класса его поля уже будут инициализированы данными. Экземпляров класса может быть столько, сколько потребуется. Если вы создали базовый класс Человек, то вы можете создать два производных объекта и назвать их Мужчина и Женщина.

По своей создание экземпляра класса очень просто. Оно сводится лишь к передаче в конструктор требуемых для работы класса данных. В данном случае наш конструктор принимает лишь текстовую строку.

Создание экземпляра класса выглядит следующим образом:

```
    +------------------ Тип класса (у типа класса первая буква
    |                   всегда должна быть заглавной)
    |
    |      +---- Имя экземпляра  класса (всегда со строковой)
    |      |
    v      v
MyClass myClass = new MyClass("some data");
                  ^              ^
                  |              +- Данные, которыми мы
                  |                 инициализируем поля класса
                  |
                  +- Ключевое слово new создает экземпляр
               класса, инициализируя его данными из конструктора
```

По умолчанию поля класса являются публичными, но прямой доступ к ним выполнять крайне не рекомендуется. Следует отметить, что далеко не обязательно поля класса должны инициализироваться из конструктора. Они так же могут инициализироваться из методов класса.

Все классы в D имеют ссылочную структуру, т.е. если мы отправляем данные в класс, мы отправляем туда копию данных, но при этом мы не создаем копию класса. Экземпляр лишь ссылается на базовый класс. Ключевое слово `new` служит именно этой задаче. Оно возвращает в экземпляр класса ссылку на базовый класс, а так же выделяет память для полей и методов.

Теперь, когда экземпляр класса создан, мы можем вызывать его методы для выполнения каких-то своих задач. Вызов метода состоит из указания имени экземпляра класса и имени самого метода.

Пример:

```
myclass.sayHello("David");
```

Т.к. наш метод тоже принимает какие-то данные в себя, то мы должны их туда отправить. Если мы этого не сделаем, то компилятор выругается: `Error: function app.MyClass.sayHello (string name) is not callable using argument types ()`. Так же он вырыругается, если мы передадим туда типы данных, которые он не ожидает. Пример:

```
myclass.sayHello(123, "David");
```

Результат:

```
Error: function app.MyClass.sayHello (string name) is not
callable using argument types (int, string)
```

В данном случае компилятор говорит нам о том, что он ожидает лишь один параметр `name` с типом `string`, а мы передаем туда два параметра `int` и `string`.

Давайте теперь посмотрим на то, как будет выглядеть законченная версия нашего приложения:

```
import std.stdio;

void main()
{
  MyClass myClass = new MyClass("some data");
  myclass.sayHello("David");
}

class MyClass
{
  string mydata;

  this(string mydata)
  {
    mydata = this.mydata;
  }

  void sayHello(string name)
  {
    writeln("Hello, ", name);
  }

  // выполняем действия над mydata
}
```

Результат:

```
> app.exe
Hello, David
```

Теперь, если нам потребуется, мы можем передавать в наш метод `sayHello` любые данные.

Пример:

```
  myclass.sayHello("David");
  myclass.sayHello("Jow");
  myclass.sayHello("Mike");
```

Результат:

```
> app.exe
Hello, David
Hello, Jow
Hello, Mike
```

Структуры

Задолго до того, как появилось ООП, в чистом Си появилось такие понятие как структуры. Структура представляет из себя некий контейнер, который так же как и сейчас объект может включать в себя поля с данными и методы работы с ними. Впоследствии, когда стала формироваться концепция ООП, от структур решено было не отказываться, т.к. их использование в ряде случаев оказалось весьма удобно. Но не смотря на то, что в ряде случаем они могут быть взаимозаменяемыми, между ними есть существенные отличия.

1. Структуры не поддерживают наследования
2. Объекты - ссылочный тип, а структуры - знаковый
3. Экземпляры класса создаются в куче, а структуры - в стеке

Данные правила кроме первого не являются законом. Если требуется, то можно не только передавать структуры по ссылке, но и размещать их в куче (используя ключевое слово `new`). Однако эти вопросы уходят очень глубоко в системное программирование и особенности работы ЭВМ.

Пример объявления структуры:

```
struct MyStruct // <-- имя структуры
{
  string name; // <-- поле структуры
  int age; // ..
  string job; // ..
}
```

Теперь для того, чтобы начать работать со структурой, необходимо создать переменную с типом этой структуры. И заполнить его данными:

```
MyStruct mystruct; // переменная mystruct с типом MyStruct
mystruct.name = "David";
mystruct.age = 20;
mystruct.job = "engineer";
```

Полный код:

```
import std.stdio;

void main()
{
  MyStruct mystruct; // создаем структуру с типом MyStruct
  mystruct.name = "David";
  mystruct.age = 20;
  mystruct.job = "engineer";

  writeln(mystruct.name);
  writeln(mystruct.age);
  writeln(mystruct.job);
  writeln(mystruct); // выводим структуру структуры
```

```
}

struct MyStruct // <-- имя структуры
{
  string name; // <-- поля структуры
  int age; // ..
  string job; // ..
}
```

Наследование

Наследование позволяет избежать дублирования кода в проекте. Оно представляет из себя способ обобщить некоторые методы, которые могут быть общими сразу в нескольких классах. К примеру, если у нас в проекте есть потребность в двух классах - в мужчине и женщине, мы можем найти между ними ряд общих моментов. К примеру, и тот и другой умееют говорить, т.е. в них должен быть реализован метод `talk`. Можно было бы конечно реализовать данный ментов в каждом отдельном экземпляре, но тогда если у нас возникла бы потребность в нем исправить, к примеру “Hello” на “Goodbye”, то правку пришлось бы делать сразу в двух местах.

Давайте для начала создадим базовый класс Человек:

```
class HumanClass
{
  string name;
  this(string name)
  {
    this.name = name;
  }

  void talk()
  {
    writeln("Hello");
  }
}
```

Наследование выглядит следующим образом. В начале вы пишите имя наследника, а потом класс, от которого хотите произвести наследование `class SubClass : MainClass`.

После чего класс потомок получает набор полей класса и методов от родителя. И далее в самом потомке вы можете реализовать уже его собственные методы.

При создании производных от базового класса классов в каждом их них требуется вызывать конструктор базового класса. За инициализацю базового конструктора отвечает слово `super`. Пример:

```
  this(string name)
    {
      super(name);
    }
```

Полный пример:

```
import std.stdio;

void main()
{
  ManClass manClass = new ManClass("Adam", 21); // создаем
экземпляр класса Мужчины
  //1. Выделится память в куче нужного размера
  //2. Вызовется конструктор ManClass.this(string name)
  //2.1. Вызовется HumanClass.this(string name) потому что
написано super()

  WomanClass womanClass = new WomanClass("Eva"); // возраст Евы
не определен и мы не передаем его в конструктор
  // создаем экземпляр класса Женщины

  manClass.talk(); // оба экземпляра класса умеют говорить
  womanClass.talk(); // оба экземпляра класса умеют говорить
  manClass.job(); // женщина умеет готовить
  womanClass.cook();

}

class HumanClass
{

  string name;
  this(string name)
  {
    this.name = name;
  }

  void talk()
  {
    writeln("Hello, ", name );
  }
}


class ManClass : HumanClass
{

  string name;
  int age;
  this(string name, int age)
  {
      super(name); // вызывем конструктор базового класса
      this.age = age; // конструктор текущего класса
  }

  void job()
  {
```

```
    writeln("Man doing job");
  }
}

class WomanClass : HumanClass
{
  string name;
  this(string name)
  {
      super(name); // вызывем конструктор базового класса
  }

  void cook()
  {
    writeln("Woman doing cooking");
  }
}
```

Интерфейсы

В C++ существовало такое понятие как множественное наследование. Т.е. была возможность создать производный класс от класса человека и насекомого, однако впоследствии эта возможность была признана ошибочной, т.к. приводила к куче пустых ошибок. В результате в современных языках программирования от нее отказались, и появилось такое понятие как интерфейсы. Интерфейсы представляют из себя лишь описание методов, которые нужно реализовать при создании самого класса. К примеру, у вас есть такие понятие как замок. Замки бывают разные, но у них есть общие правила. Замок можно открывать как с помощью ключа, так и с помощью пароля. Однако реализация самих методов открытия и закрытия может быть совершенно разной. Однако у всех них будет какое-то общее начало. Все они должны должны реализовывать такие понятия как: “Открыть” и “Закрыть”.

Сами интерфейсы не отвечают на вопрос “как делать?”, они говорят лишь, что нужно делать. А сама реализация лежит на плечах унаследованного от них класса.

Интерфейсы нужны для того, чтобы для разных сущностей был один общий интерфейс взаимодействия.

Интерфейсы созданы для единообразия. Поэтому классы унаследованные от интерфейсов должны реализовывать **все** их методы.

Чтобы визуально отличать интерфейсы от классов им принято давать имена, начинающиеся с заглавной буквы “I”. Пример: `IUsers`.

Давайте рассмотрим заготовку:

```
import std.stdio;

void main()
{
  ManClass manClass = new ManClass();

}

interface IHuman // в интерфейсе создана два метода
{
  // Данные методы должны быть реализованы во всех производных
классах
  void talk();
  void walk();
}


class ManClass : IHuman
{
  void talk()
  {

  }

  void walk()
  {

  }

}

class WomanClass : IHuman
{

  void talk()
  {

  }

  void walk()
  {

  }

}
```

Как видно в каждом классе, унаследованном от интерфейса, мы реализовали все методы интерфейсов.

Полный пример:

```
import std.stdio;

void main()
{
  ManClass manClass = new ManClass("David"); // передаем данные
 в конструктор класса
  manClass.talk(); // вызываем метод класса
  manClass.walk("David"); // передаем данные в метод напрямую

}

interface IHuman // в интерфейсе создана два метода
{
  // Данные методы должны быть реализованы во всех производных
классах
  void talk();
  void walk(string manname); // если реализованный метод должен
принимать данные, то и метод интерфейса должен их принимать
}


class ManClass : IHuman
{
  string name;
  this(string name)
  {
    this.name = name;
  }

  void talk()
  {
    writefln("%s can talk", name); // используем данные из
конструктора
  }

  void walk(string manname) // принимаем данные переданные в
метод
  {
    writefln("%s can walk", manname);
  }

}

class WomanClass : IHuman
{

  void talk()
  {
    // пока не реализовано
  }
```

```
  void walk(string womanname) // тоже принимает данные, т.к. в
реализации
  {
    // пока не реализовано
  }

}
```

Результат:

```
> app.exe
David can talk
David can walk
```

Абстрактные методы и абстрактные классы

Бывают ситуации, когда класс может содержать в себе как реализованные методы, так и их объявление (такое, которое содержится в интерфейсах). Если в классе есть хотя бы один незавершенный метод, то такой класс называется абстрактным. Объявление подобного метода начинается со слова `abstract`.

Класс так же может быть объявлен с ключевым словом `abstract`. Экземпляр подобного класса можно создать только унаследовавшись от него.

При реализации в производном классе такой класс должен быть перегружен оператором `override`.

Пример:

```
import std.stdio;

void main()
{
  ManClass manClass = new ManClass("David");
  manClass.talk(); // вызываем завершенный метод
  manClass.sayHello("Jow"); // в начале реализуем, а потом
вызываем
}

class Human
{
  abstract void sayHello(string name); // метод не завершен и
его нужно реализовать
  void talk() // метод завершен и его можно только вызывать
  {
    writeln("I can talk");
  }
}



class ManClass : Human
```

```
{
  string name;
  this(string name)
  {
    this.name = name;
  }

  override void sayHello(string somename)
  {
    writeln("Hello, ", somename);
  }

}
```

Результат:

```
> app.exe
I can talk
Hello, Jow
```

Шаблонные функции

Шаблонные функции (templates) позволяют компилятору автоматически генерировать код в зависимости от того, какой тип данных передается в функцию. Значительно количество функций в D выполнено в виде шаблонов.

Простая функция на вход может принять только один (указанный в ее аргументах) тип данных. К примеру:

```
void foo(int x)
{

}
```

Бывают нужно, чтобы одна функция могла принимать на вход различные типы данных. К примеру, функция `writeln()` может принимать как числа, так и строки.

Если требуется, мы можем выполнить перегрузку функции, описав ее с тем же именем, но другими входящими аргументами, к примеру:

```
void foo(string x)
{

}
```

Многозадачность

Потоки и файберы

Изначально процессор занимается последовательным выполнением поступающих в него команд и ничего не знает ни о процессах, ни о потоках. Для того, чтобы мы могли работать с несколькими приложениями одновременно операционная система реализует для нас механизм многозадачности.

Его идея крайне простая - заставить каждое приложение думать, что на процессоре в настоящий момент запущено только оно одно. Но при этом каждому отдельному приложению давать доступ к процессору лишь на очень небольшой квант времени, после чего выполнять операцию переключение контекста т.е. брать полный слепок текущего состояния процессора, сохранять его в оперативной памяти и загружать на процессор данные нового приложения. За счет того, что переключение между процессами происходит очень быстро, пользователю кажется, что все приложения на компьютере выполняются одновременно.

Главную сложность в многопоточном программировании представляет из себя работа с разделяемыми данными т.е. такими данными, которые может модифицировать сразу несколько потоков. Однако с точки зрения программиста самый удобный код это код который исполняется линейно и последовательно. Как только эта линейность нарушается в программах начинает появляться большое количество различных ошибок.

Именно поэтому первым типом многозадачности была кооперативная многозадачность. Ее идея была в следующем - давать каждому приложению использовать процессор столько времени сколько потребуется, после чего каждое приложение должно было самостоятельно передать управление другому приложению. Однако любая ошибка в приложении или даже задержка при обращении к каким-либо данным приводили к тому, что приложение не передавало управление процессором дальше и с точки зрения пользователя это выглядело как зависание системы.

Поэтому очень скоро от этой практики решено было оказаться и задачу по переключению приложений возложили на операционную систему. Такой тип многозадачности был назван вытесняющим. Вот тут то уже переключение контекста происходило через определенный квант времени. Однако и этот подход имел свои недостатки.

Переключение контекста является достаточно время затратной операцией т.к. требует не только сохранения состояния целого ряда регистров, но и обращения к оперативной памяти, которая в несколько раз медленнее, чем регистры и кэш CPU.

Если на процессоре исполняется лишь несколько приложений, то это не приводит к заметным просадкам производительности, но с ростом количества одновременно исполняемых задач все больше и больше времени будет уходить исключительно на саму операцию переключение контекста в результате, к примеру, если у вас будет запущено 1000 потоков, то 90% времени будет уходить исключительно на переключение между ними.

Но это еще не все. Не все потоки выполняют на процессоре какие-либо вычисления. К примеру, если вы пишете веб-сервер, то большая часть времени у потока будет уходить на ожидание ответа от базы данных или файловой системы, а значит поток все это время будет простаивать.

В случае с десктопными приложениями подобная проблема возникает крайне редко, но для сетевых приложений ситуация иная. Большинство веб-серверов обслуживает сотни и тысячи подключений одновременно. Это значит, что каждое новое подключение мало того, что будет требовать создание нового потока обработки, так еще и сам поток будет 90% времени заниматься ожиданием.

Для решения этой проблемы в D была введена концепция файберов (fibers). Файберы представляют из себя потоки, работающие поверх системных потоков в юзеспейсе и при этом самостоятельно управляющие временем своей жизни. Идея была в следующем.

Если системные потоки стоят крайне дорого и временем их жизни управлять крайне сложно (хотя и можно), то файберы должны взять на себя все блокирующие операции т.е. такие операции, которые не требуют каких-либо рассчетов, а большую часть времени у них уходит на ожидание ввода-вывода (обычно это файловая система и база данных).

В задачу файбера входит лишь какая-то операция к примеру, отправить запрос на сетевое устройство и не дожидаясь ответа передать управление другому файберу.

Вы можете спросить, но как определить какие операции являются блокирующими, а какие нет. Ведь с практической точки зрения любой рассчет на CPU будет точно таким же блокирующим т.к. без его выполнения невозможно продолжить дальнейшую работу.

Это действительно так и грань между тем где лучше использовать файберы, а где системные потоки является очень тонкой. Однако главное правило заключается в том, что **любые расчетные операции следует производить в системных потоках**, а **любые операции которые требуют ожидания завершения операции ввода-вывода в файберах**.

Тут следует отметить, что Файберы это библиотечная функция и конкретная их реализация зависит исключительно от того какой именно библиотекой вы пользуетесь. В настоящий момент в D есть две реализации файберов. Первая в стандартной библиотеке Phobos, вторая в фреймворке vibed.

Для себя файберы проще всего представлять как функции, которые выполняются последовательно одна за другой. При этом у них отсутствют проблемы с общими данными и очередностью обращения к ним.

Все файберы встают на выполнение в очередь `event loop`, откуда они запускаются один за другим.

Но прежде чем говорить о файберах необходимо сказать несколько слов про итераторы и генераторы. Именно на идее генераторов и построена сама концепция файберов.

Если итератор ходит по некому контейнеру, содержимое которого уже посчитано и лежит в памяти, то генератор представляют из себя диапазон (InputRange) который выплевывает из себя значения одно за другим, высчитывая следующее по каким-то правилам (при этом в памяти не хранит ни прошлые, ни следующие). При этом когда нам нужно пробежаться по какому-то диапазону значений и что-то с ними поделать - для нас нет разницы как они получены (из контейнера или из генератора). Поэтому там, где получается использовать генератор - это эффективно.

Для того, чтобы заставить генератор отдать значение используется ключевое слово `yield`. `yield` по своей сути очень похож на оператор `return`, только в отличие от последнего `yield` не выполняет выход из функции, а просто приостанавливает ее работу. `yield` может использоваться только внутри файбера т.к. простые функции про файбер ничего не знают.

Пример создания файбера и передача ему в качестве аргумента вызываемой функции:

```
auto f = new Fiber(&foo);
```

`f.call();` вызов файбера
`Fiber.yield();` метод `yield()` класса `Fiber` вызывающий приостановку выполнение текущей функции

Пример:

```
import std.stdio;
import core.thread;

void main()
{
  auto f = new Fiber(&foo);
  f.call(); // Prints Hello
  f.call(); // Prints World
}

void foo()
{
  writeln("Hello");
  Fiber.yield();
  writeln("World");
}
```

Результат:

```
> app.exe
Hello
World
```

Как видно из кода мы в начале создаем файбер передавая туда вызываемую функцию, потом запускаем его методом `call`, потом прерываем выполнение данной функции `Fiber.yield();` и затем снова делаем `call` чтобы продолжить ее выполнение с того момента когда она остановилась.

Реализация файберов в `vibed` очень похожа. И большинство функций во фреймворке автоматически выполняют `yield` в процессе своей работы.

Пример асинхронной функции чтения файла:

```
  alias read = Stream.read;
  size_t read(scope ubyte[] dst, IOMode)
  {
    assert(this.readable);
    size_t len = dst.length;
    while (dst.length > 0) {
      enforce(dst.length <= leastSize);
      auto sz = min(dst.length, 4096);
      enforce(() @trusted { return .read(m_fileDescriptor,
dst.ptr, cast(int)sz); } () == sz, "Failed to read data from
disk.");
      dst = dst[sz .. $];
      m_ptr += sz;
      yield();
    }
    return len;
}
```